NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE
SOFTWARE ENGINEERING LAB

Investigation of AI tools Performance in the Definition of Microservices Software Architectures

Diploma Thesis
Of
GEORGIOS SOTIROPOULOS

Supervisor: Vassilios Vescoukis, Professor, NTUA

Athens, July 2025

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE
SOFTWARE ENGINEERING LAB

Investigation of AI tools Performance in the Definition of Microservices Software Architectures

Diploma Thesis
Of

GEORGIOS SOTIROPOULOS

Supervisor: Vassilios Vescoukis, Professor, NTUA

Approved by the examination committee on the 4th July 2025.

V.Vescoukis	G.Stamou	N.Papaspyrou
Professor, NTUA	Professor, NTUA	Professor, NTUA

Athens, July 2025

.....
Sotiropoulos Georgios

Graduate of School of Electrical and Computer Engineering, National Technical University of Athens

Περίληψη

Ο σχεδιασμός της αρχιτεκτονικής λογισμικού αποτελεί ένα καθοριστικό βήμα στον κύκλο ζωής ανάπτυξης λογισμικού, γεφυρώνοντας τις απαιτήσεις του λογισμικού με την υλοποίηση του συστήματος μέσω του καθορισμού δομών υψηλού επιπέδου. Ο σχεδιασμός της αρχιτεκτονικής παραμένει μια απαιτητική, χρονοβόρα και επιρρεπής σε σφάλματα διαδικασία. Η παρούσα εργασία εξετάζει την απόδοση εργαλείων τεχνητής νοημοσύνης (AI), και συγκεκριμένα των Large Language Models (LLMs), στην αυτόματη παραγωγή αρχιτεκτονικών λογισμικού με έμφαση σε συστήματα βασισμένα σε Microservices. Βασιζόμενη σε προηγούμενη έρευνα, η μελέτη αυτή διερευνά πώς διαφορετικές μορφές εισόδου, από απλό κείμενο απαιτήσεων έως λεπτομερή έγγραφα προδιαγραφών (SRS), η επιλογή μοντέλου και οι τεχνικές Retrieval-Augmented Generation (RAG), επηρεάζουν την ποιότητα και τη συμμόρφωση των αρχιτεκτονικών σχεδιασμών που παράγονται από την τεχνητή νοημοσύνη σε σχέση με τις απαιτήσεις του λογισμικού. Εφαρμόζουμε ένα πλαίσιο αξιολόγησης βασισμένο σε εκτιμήσεις ειδικών του πεδίου και εισάγουμε ένα σύνολο αντικειμενικών ποσοτικοποιημένων δεικτών με στόχο τη μετάβαση σε μια αυτόματη διαδικασία αξιολόγησης. Επιπλέον, διερευνάται εάν μικρότερα, τοπικά φιλοξενούμενα LLMs μπορούν να αποτελέσουν πρακτικές εναλλακτικές λύσεις σε εμπορικά διαθέσιμα εργαλεία AI. Τα αποτελέσματα προσφέρουν σημαντικές γνώσεις για τη δυνατότητα της τεχνητής νοημοσύνης να βοηθήσει στο στάδιο του αρχιτεκτονικού σχεδιασμού στην ανάπτυξη λογισμικού, ενισχύοντας την αποδοτικότητα και την ποιότητα του σχεδιασμού, ενώ παράλληλα επαναπροσδιορίζεται ο ρόλος των μηχανικών λογισμικού σε συνεργατικά περιβάλλοντα ανθρώπου – τεχνητής νοημοσύνης. Η εργασία αυτή συμβάλλει στον αναδυόμενο τομέα της υποβοηθούμενης από AI, τεχνολογίας λογισμικού και σκιαγραφεί μελλοντικές ερευνητικές κατευθύνσεις για την περαιτέρω ενσωμάτωση της αυτοματοποίησης στον σχεδιασμό πολύπλοκων συστημάτων.

Λέξεις Κλειδιά: Large Language Models (LLMs), αρχιτεκτονική λογισμικού, τεχνολογία λογισμικού, τεχνητή νοημοσύνη (AI), Retrieval-Augmented Generation (RAG), UML, Microservices, Software Requirements Specification (SRS).

Abstract

The design of software architecture is a pivotal step in the software development lifecycle, bridging user requirements and system implementation through the definition of high-level structural designs. Despite its critical importance, architectural design remains a challenging, time-intensive, and error-prone process. This thesis investigates the performance of artificial intelligence (AI) tools, specifically large language models (LLMs), in automating the generation of software architectures focusing on Microservices-based systems. Building on prior research, this study explores how different input formats, ranging from plain-text requirements to detailed specification documents, model selection and Retrieval-Augmented Generation (RAG) techniques, affect the quality and compliance of AI-generated architectural designs with the software requirements. We apply an evaluation framework based on assessments from domain experts and we introduce a set of objective metrics to pave the road towards an automatic evaluation process. Additionally, this study explores whether smaller, locally hosted LLMs can serve as practical alternatives to commercially available AI tools. The results provide insights into the potential for AI to transform the architectural design phase of software development, enhancing design efficiency and quality while reshaping the role of software architects in collaborative human-AI workflows. This work contributes to the growing field of AI-assisted software engineering and outlines future research avenues to further integrate intelligent automation into complex system design.

Keywords: Large Language Models (LLMs), software engineering, software architecture, artificial intelligence (AI), Retrieval-Augmented Generation (RAG), UML, Microservices, Software Requirements Specification (SRS).

Acknowledgements

I would like to express my sincere gratitude to my Professor, Mr. Vassilios Vescoukis, for his valuable guidance, support, and constructive feedback throughout this thesis. His expertise was crucial in shaping the direction and outcome of my work.

I am also grateful to PhD Researcher Mr. Christos Hadjichristofi, whose advice, assistance and ongoing support played a key role in the development of this thesis.

As this diploma thesis marks the completion of my studies at the School of Electrical and Computer Engineering at the National Technical University of Athens (NTUA), I would like to extend my thanks to my family and friends. Their continuous support and encouragement have been with me every step of the way, making this journey both meaningful and fulfilling.

Finally, I would like to thank all the teaching staff at NTUA for sharing their knowledge and promoting a motivating and inspiring academic environment throughout these years.

Table of Contents

Table of Figures

GREK EXTENDED ABSTRACT

Ο κύκλος ζωής της ανάπτυξης λογισμικού είναι μια πολύπλοκη διαδικασία πολλών σταδίων, που εκτείνεται από τη συλλογή απαιτήσεων του λογισμικού έως την ανάπτυξη και συντήρηση πλήρως λειτουργικών και σύνθετων συστημάτων. Περιλαμβάνει βασικές φάσεις όπως ανάλυση απαιτήσεων, σχεδιασμό αρχιτεκτονικής, υλοποίηση, δοκιμές, εγκατάσταση και συντήρηση.

Το στάδιο του σχεδιασμού της αρχιτεκτονικής του συστήματος αποτελεί κρίσιμο το σημείο μετάβασης από τις αφηρημένες απαιτήσεις στην πρακτική υλοποίηση. Σε αυτό το στάδιο, οι μηχανικοί λογισμικού ορίζουν τις βασικές δομές και σχέσεις που θα αποτελέσουν τον σκελετό του συστήματος, δημιουργώντας μοντέλα (συχνά με χρήση της γλώσσας UML) τα οποία λειτουργούν ως καθοδηγητικά σχέδια για όλες τις ομάδες ανάπτυξης. Η αρχιτεκτονική δεν ικανοποιεί μόνο τις λειτουργικές απαιτήσεις, αλλά και μη λειτουργικές όπως η επεκτασιμότητα και η απόδοση. Μία σωστή αρχιτεκτονική μειώνει τα λάθη, θέτει σαφή τεχνικά πρότυπα και παρέχει σταθερή βάση για μελλοντική εξέλιξη. Ωστόσο, ο σχεδιασμός της αρχιτεκτονικής και η δημιουργία των αντίστοιχων διαγραμμάτων είναι μια χρονοβόρα και ευάλωτη σε σφάλματα διαδικασία, συχνά λόγω παρερμηνειών αρχιτεκτονικών αρχών, με αποτέλεσμα καθυστερήσεις και αναποτελεσματικότητα σε όλη τη διαδικασία ανάπτυξης.

Η ραγδαία πρόοδος της τεχνητής νοημοσύνης, και ιδιαίτερα των μεγάλων γλωσσικών μοντέλων (LLMs) βασισμένων στη βαθιά μάθηση, έχει ανοίξει νέες δυνατότητες αυτοματοποίησης εργασιών που μέχρι πρόσφατα απαιτούσαν ανθρώπινη παρέμβαση. Αυτά τα μοντέλα, που εκπαιδεύονται σε τεράστιους όγκους δεδομένων, έχουν δείξει υψηλή ικανότητα κατανόησης και παραγωγής φυσικής γλώσσας. Ήδη αξιοποιούνται για εντοπισμό σφαλμάτων και δημιουργία κώδικα, γεγονός που εγείρει το ερώτημα: μπορούν να χρησιμοποιηθούν και για την αυτοματοποιημένη παραγωγή αρχιτεκτονικών σχεδίων και UML διαγραμμάτων; Εάν μπορούν να μετατρέψουν γραπτές απαιτήσεις σε οργανωμένα, συνεπή αρχιτεκτονικά μοντέλα σύμφωνα με συγκεκριμένα πρότυπα, θα λύσουν μια από τις πιο απαιτητικές φάσεις του σχεδιασμού λογισμικού.

Η παρούσα μελέτη εξετάζει την ικανότητα των LLMs να παράγουν αρχιτεκτονικές λογισμικού, σε μορφή διαγραμμάτων UML, με έμφαση στα Microservices. Χτίζοντας πάνω σε προηγούμενη έρευνα, δοκιμάζουμε διάφορες μορφές εισόδου (από απλές περιγραφές απαιτήσεων μέχρι πλήρη έγγραφα Software Requirements Specification) αξιολογώντας πώς το εκάστοτε μοντέλο και η χρήση τεχνικών ενίσχυσης ανάκτησης γνώσης (Retrieval-Augmented Generation) επηρεάζουν την ποιότητα των παραγόμενων σχεδίων.

Κεντρικό ερώτημα αποτελεί το κατά πόσο οι παραγόμενες αρχιτεκτονικές ικανοποιούν πλήρως τις απαιτήσεις και τηρούν τις αρχές σχεδίασης. Εισάγουμε σύνολο κριτηρίων αξιολόγησης για ανθρώπινη αξιολόγηση και προτείνουμε ποσοτικοποιημένους δείκτες αντικειμενικής αξιολόγησης βασισμένες σε προηγούμενες έρευνες. Παράλληλα, εξετάζουμε εάν

μικρότερα, τοπικά εκτελούμενα LLMs μπορούν να αποτελέσουν βιώσιμες εναλλακτικές έναντι εμπορικών λύσεων.
Διερευνώντας τις δυνατότητες και τους περιορισμούς της τεχνητής νοημοσύνης στον σχεδιασμό αρχιτεκτονικών λογισμικού, η παρούσα έρευνα επικεντρώνεται στα εξής ερευνητικά ερωτήματα:

1. **Ποια μορφή εξόδου προσφέρει τη βέλτιστη απεικόνιση αρχιτεκτονικών από LLMs;**
   Διερευνούμε ποια μορφή αναπαράστασης (όπως XMI, εικόνες, PlantUML ή Mermaid διαγράμματα) επιτρέπει στα LLMs να παράγουν αρχιτεκτονικά σχέδια υψηλής ποιότητας και ακρίβειας.

2. **Βελτιώνει ένα δομημένο έγγραφο SRS την ποιότητα των παραγώμενων διαγραμμάτων από τα LLMs;**
   Συγκρίνουμε τα αποτελέσματα των μοντέλων όταν λαμβάνουν ως είσοδο πλήρες έγγραφο Software Requirements Specification σε σχέση με απλές λίστες απαιτήσεων. Εξετάζουμε κατά πόσο μπορούν να κατανοήσουν και να μεταφράσουν σύνθετη επιχειρησιακή λογική και διασυνδεόμενες απαιτήσεις σε συνεκτικό αρχιτεκτονικό σχεδιασμό.

3. **Μπορούν τα LLMs να παράγουν αρχιτεκτονικές Microservices για πιο σύνθετες εφαρμογές;**
   Ελέγχουμε την ικανότητα των LLMs να διασπούν σύνθετες απαιτήσεις σε κατάλληλες μικρο-υπηρεσίες (microservices), τηρώντας τις αρχές σχεδίασης, όπως τα όρια υπηρεσιών, οι σχέσεις μεταξύ τους και τα πρότυπα επικοινωνίας.

4. **Ποια κριτήρια αξιολόγησης είναι καταλληλότερα για αρχιτεκτονικές Microservices;**
   Εντοπίζουμε τα κενά των υφιστάμενων μεθόδων αξιολόγησης και αναπτύσσουμε εξειδικευμένα κριτήρια που ανταποκρίνονται στις ιδιαιτερότητες του σχεδιασμού Microservices.

5. **Πώς μπορούμε να υπολογίσουμε αντικειμενικούς δείκτες αξιολόγησης για τις παραγόμενες αρχιτεκτονικές;**
   Με στόχο την αξιόπιστη αξιολόγηση της ποιότητας των παραγόμενων αρχιτεκτονικών, προτείνουμε ποσοτικοποιημένους δείκτες που εισάγουν μία αντικειμενική διάσταση στην αξιολόγηση της ποιότητας των παραγόμενων διαγραμμάτων.

6. **Παρουσιάζουν τα LLMs «ψευδαισθήσεις» όταν υπολογίζουν τους δείκτες μέτρησης για τα δικά τους διαγράμματα κλάσεων;**
   Εξετάζουμε την ακρίβεια των υπολογισμών που κάνουν τα μοντέλα όταν καλούνται να υπολογίσουν τους δείκτες μέτρησης για τα διαγράμματα που τα ίδια παρείχαν, εντοπίζοντας τυχόν σφάλματα κατανόησης ή αυθαίρετες απαντήσεις.

7. **Πώς επηρεάζει η τεχνική Retrieval-Augmented Generation (RAG) την ποιότητα των διαγραμμάτων;**
   Ερευνάμε εάν η αξιοποίηση εξωτερικού πληροφοριακού περιεχομένου μέσω RAG ενισχύει την ακρίβεια, τη συνέπεια και τη βαθύτερη κατανόηση του πλαισίου από το μοντέλο.

8. **Πώς επηρεάζει η μία δεύτερη ερώτηση στα LLMs την ποιότητα του αρχιτεκτονικού σχεδιασμού;**
   Μελετάμε εάν διαδοχικές αλληλεπιδράσεις, χωρίς προσθήκη εξωτερικής γνώσης, οδηγούν σε προοδευτικά βελτιωμένο σχεδιασμό. Αξιολογούμε τη συνεισφορά της επαναληπτικής βελτίωσης ως προς την πληρότητα, την ακρίβεια και τη συνοχή του τελικού αποτελέσματος.

Προκειμένου να προσπαθήσουμε να απαντήσουμε στα παραπάνω ερευνητικά ερωτήματα, σχεδιάσαμε και υλοποιήσαμε μια σειρά πειραμάτων ώστε να εντοπίσουμε τη διαφοροποίηση στην ποιότητα των παραγόμενων αρχιτεκτονικών διαγραμμάτων.

Οι παράμετροι που τροποποιήθηκαν:

- **Το υπό ανάπτυξη λογισμικό:**
  DCC (λογισμικό με περιορισμένες και απλές απαιτήσεις) και MyCharts (πιο σύνθετο λογισμικό με αυξημένο πλήθος και πολυπλοκότητα απαιτήσεων).
- **Η μορφή παρουσίασης των απαιτήσεων προς τα LLMs:**
  Είτε ως απλή λίστα λειτουργικών και μη λειτουργικών απαιτήσεων, είτε ως πλήρως δομημένο έγγραφο Software Requirements Specification (SRS).
- **Η ζητούμενη αρχιτεκτονική προσέγγιση:**
  Client-Server, Three-Tier, MVC ή Microservices.
- **Το γλωσσικό μοντέλο (LLM) που χρησιμοποιήθηκε για κάθε δοκιμή.**
- **Η χρήση της τεχνικής Retrieval-Augmented Generation (RAG):**
  Αν χρησιμοποιήθηκε ή όχι, καθώς και ποιο συγκεκριμένο αρχείο RAG αξιοποιήθηκε σε κάθε περίπτωση.

Τα πειράματα που υλοποιήθηκαν και ο σκοπός τους:

1. **Πείραμα DCC – Ανασκόπηση**
   Εξετάζει κατά πόσο τα LLMs αποδίδουν καλύτερα όταν λαμβάνουν ως είσοδο ένα δομημένο έγγραφο Προδιαγραφών Απαιτήσεων Λογισμικού (SRS), σε σύγκριση με απλές λίστες λειτουργικών και μη λειτουργικών απαιτήσεων. Για τον σκοπό αυτό, αναπαράγεται το αρχικό πείραμα της δουλειάς στην οποία βασιστήκαμε, αυτή τη φορά με χρήση SRS.

2. **Πείραμα MyCharts**
   Στόχος του πειράματος είναι να αξιολογηθεί η ικανότητα των LLMs στο να σχεδιάζουν αρχιτεκτονικές για μια πιο σύνθετη εφαρμογή, βασισμένη στο πρότυπο αρχιτεκτονικής Microservices. Το πείραμα αυτό αποτελεί την κύρια εστίαση της μελέτης.

3. **Πείραμα MyCharts με 2 Διαδοχικά Prompts**
   Σε αυτό το πείραμα εξετάζουμε πειραματικά εάν η χρήση ενός δεύτερου, δομημένου prompt προς το LLM μπορεί να βελτιώσει την ποιότητα των παραγόμενων διαγραμμάτων.

Σε αυτό το σημείο, παρουσιάζουμε συνοπτικά την ανάλυση των επιμέρους πειραμάτων:

Πείραμα DCC – Ανασκόπηση

Το πείραμα DCC επικεντρώνεται στην αξιολόγηση της επίδρασης που έχει η μορφή παρουσίασης των απαιτήσεων στην απόδοση των Μεγάλων Γλωσσικών Μοντέλων (LLMs) κατά τη δημιουργία αρχιτεκτονικών λογισμικού. Η εφαρμογή που χρησιμοποιήθηκε, με την ονομασία Dummy Coordination Conversion (DCC), είναι ένα σχετικά απλό σύστημα διαχείρισης συντεταγμένων σε καρτεσιανή και πολική μορφή, το οποίο επιτρέπει στους χρήστες να μετατρέπουν, αποθηκεύουν, ανακτούν, τροποποιούν και διαγράφουν ομάδες συντεταγμένων.

Το πείραμα υλοποιήθηκε εξετάζοντας τρεις διαφορετικές αρχιτεκτονικές προσεγγίσεις: Client-Server, Three-Tier και Model-View-Controller (MVC). Ο στόχος ήταν να διαπιστωθεί εάν τα LLMs μπορούν να αποδώσουν καλύτερες αρχιτεκτονικές λύσεις όταν τους παρέχεται ένα δομημένο έγγραφο Προδιαγραφών Απαιτήσεων Λογισμικού (SRS), σε σύγκριση με απλές λίστες λειτουργικών και μη λειτουργικών απαιτήσεων (FR-NFR lists).

Η βασική δομή του prompt που χρησιμοποιήθηκε ήταν ίδια με αυτή που είχε εφαρμοστεί σε προηγούμενο πείραμα προηγούμενης έρευνας, με τη μοναδική διαφοροποίηση να εντοπίζεται στην αντικατάσταση των λιστών απαιτήσεων με SRS έγγραφα. Χρησιμοποιήθηκε προαιρετικά RAG (Retrieval-Augmented Generation) με υλικό από το βιβλίο «Software Engineering» (10η έκδοση) του Ian Sommerville (ίδιο RAG αρχείο σε σύγκριση με προηγούμενο πείραμα).

Το πείραμα πραγματοποιήθηκε με δύο εκδοχές του εγγράφου SRS: μια συνοπτική και μία εκτενέστερη, με στόχο να διαπιστωθεί αν το επίπεδο λεπτομέρειας στο έγγραφο επηρεάζει την ποιότητα και την ακρίβεια των παραγόμενων αρχιτεκτονικών διαγραμμάτων.

Για την αξιολόγηση των παραγόμενων διαγραμμάτων διατηρήθηκαν τα ίδια κριτήρια με το προηγούμενο πείραμα, ώστε να εξασφαλιστεί η συνέπεια και η δυνατότητα έγκυρης σύγκρισης των αποτελεσμάτων. Κάθε διάγραμμα βαθμολογήθηκε από 0 έως 5 με βάση τα παρακάτω:

1. **Συμμόρφωση με την αιτούμενη αρχιτεκτονική**
   Εξετάστηκε κατά πόσο το παραγόμενο διάγραμμα υλοποιεί σωστά την αρχιτεκτονική που ζητήθηκε (όπως Client-Server, MVC κ.λπ.). Δόθηκε έμφαση στην ορθή εφαρμογή των αρχιτεκτονικών αρχών και στην ορθή κατανομή των ευθυνών μεταξύ των κλάσεων.

2. **Ορθότητα των σχέσεων μεταξύ κλάσεων**
   Αξιολογήθηκε η ακρίβεια των σχέσεων (π.χ. συσχετίσεις, εξαρτήσεις, κληρονομήσεις) μεταξύ των κλάσεων, στο πλαίσιο της αρχιτεκτονικής που ζητήθηκε.

3. **Συνοχή και Συζευξιμότητα**
   Οι αξιολογητές έκριναν το κάθε διάγραμμα ως προς το βαθμό συνοχής (δηλαδή κατά πόσο κάθε κλάση έχει ενιαίο και σαφώς καθορισμένο σκοπό) και τη συζευξιμότητα (κατά πόσο

ελαχιστοποιούνται οι εξαρτήσεις μεταξύ κλάσεων), επιδιώκοντας υψηλή συνοχή και χαμηλή συζευξιμότητα.

4. **Συνέπεια με τις απαιτήσεις του λογισμικού**
   Αξιολογήθηκε κατά πόσο το διάγραμμα ανταποκρίνεται στις λειτουργικές και μη λειτουργικές απαιτήσεις που είχαν δοθεί, είτε με τη μορφή λιστών είτε μέσω SRS εγγράφων.

Αφού ολοκληρώθηκε η παραγωγή των διαγραμμάτων από τα LLMs, ακολούθησε η αξιολόγησή τους βάσει των προαναφερθέντων κριτηρίων. Τα βασικά ευρήματα σχετικά με την ποιότητα των παραγόμενων διαγραμμάτων συνοψίζονται ως εξής:

1. Τα μεγαλύτερα εμπορικά μοντέλα απέδωσαν γενικά καλύτερα από τα μικρότερα, τοπικά μοντέλα, δημιουργώντας πιο δομημένα και συνεπή διαγράμματα.

2. Η χρήση RAG οδήγησε σε μεικτά αποτελέσματα, ιδιαίτερα όταν συνδυάστηκε με εκτενείς εισόδους από πλήρη έγγραφα SRS. Στα μεγάλα μοντέλα πολλών παραμέτρων η επιπλέον πληροφορία λειτούργησε υποστηρικτικά, ενώ σε μικρότερα μοντέλα προκάλεσε σύγχυση και μείωση της ποιότητας.

3. Κατά τη σύγκριση μεταξύ αναλυτικών και συνοπτικών εκδοχών του SRS, η απόδοση των μοντέλων παρέμεινε σχετικά σταθερή, γεγονός που υποδηλώνει πως οι πιο εκτενείς περιγραφές δεν οδηγούν απαραίτητα σε καλύτερης ποιότητας διαγράμματα.

4. Όσον αφορά τη σύγκριση μεταξύ πλήρων SRS εγγράφων και απλών λιστών λειτουργικών και μη λειτουργικών απαιτήσεων (FR-NFR), τα μεγαλύτερα μοντέλα ανταποκρίθηκαν καλύτερα στα SRS, δημιουργώντας διαγράμματα υψηλότερης ποιότητας. Αντίθετα, τα μικρότερα μοντέλα φάνηκε να δυσκολεύονται με το εκτενές περιεχόμενο και απέδωσαν καλύτερα όταν τους δόθηκαν οι πιο σύντομες και στοχευμένες λίστες.

5. Ο τύπος της ζητούμενης αρχιτεκτονικής επηρέασε επίσης την ποιότητα των παραγόμενων διαγραμμάτων. Ιδιαίτερα στην περίπτωση της αρχιτεκτονικής Client-Server, τα αποτελέσματα αξιολογήθηκαν χαμηλότερα, ενώ αντίθετα, η αρχιτεκτονική Three-Tier οδήγησε σε υψηλότερες βαθμολογίες.

Η πλήρης περιγραφή του πειράματος, με αναλυτική παρουσίαση όλων των παραμέτρων και των αποτελεσμάτων, παρατίθεται στο κυρίως αγγλικό κείμενο που ακολουθεί την παρούσα ελληνική περίληψη.

Πείραμα MyCharts

Στο πείραμα MyCharts, ο στόχος ήταν να διερευνηθεί η ικανότητα των μεγάλων γλωσσικών μοντέλων (LLMs) να σχεδιάζουν αρχιτεκτονικές λογισμικού για εφαρμογές αυξημένης πολυπλοκότητας, βασισμένες στο αρχιτεκτονικό πρότυπο των Microservices.

Η εφαρμογή MyCharts είναι μια διαδικτυακή πλατφόρμα που απευθύνεται κυρίως σε μη τεχνικούς χρήστες και έχει σχεδιαστεί με στόχο να απλοποιεί τη δημιουργία γραφημάτων. Παρέχει τη δυνατότητα στον χρήστη να κατεβάζει έτοιμα πρότυπα CSV για συγκεκριμένους τύπους γραφημάτων, να ανεβάζει τα δικά του δεδομένα, και να δημιουργεί αυτόματα γραφήματα μέσω της βιβλιοθήκης Highcharts. Τα γραφήματα μπορούν στη συνέχεια να αποθηκευτούν ή να ληφθούν σε διάφορες μορφές, όπως PDF, PNG, SVG και HTML. Επιπλέον, ο χρήστης μπορεί να αγοράσει "πακέτα" χρήσης (quotas) για να δημιουργήσει περισσότερα γραφήματα, καθώς και να προβάλλει ή να κατεβάσει τα ήδη δημιουργημένα.

Η αρχιτεκτονική που ζητήθηκε από τα μοντέλα ήταν Microservices, γεγονός που αυξάνει σημαντικά τον βαθμό δυσκολίας, καθώς απαιτεί σωστή κατανομή λειτουργιών σε ανεξάρτητες υπηρεσίες, διαχείριση αλληλεπιδράσεων μεταξύ των υπηρεσιών και σωστή απομόνωση ευθυνών.

Στο πλαίσιο του πειράματος χρησιμοποιήθηκαν και υλικά υποστήριξης τύπου RAG (Retrieval-Augmented Generation) για ορισμένες δοκιμές. Συγκεκριμένα, αξιοποιήθηκαν αποσπάσματα από τα βιβλία Microservices Patterns του Chris Richardson (Κεφάλαιο 2) και Microservices Design Patterns του Nishant Malhotra.

Τα μοντέλα κλήθηκαν να παράγουν αρχιτεκτονικά διαγράμματα βασισμένα σε δύο διαφορετικά είδη εισόδου περιγραφής των απαιτήσεων του λογισμικού: σύντομες λίστες λειτουργικών και μη λειτουργικών απαιτήσεων (FR/NFR), ή ένα πληρέστερο έγγραφο προδιαγραφών λογισμικού (SRS).

Για την αξιολόγηση των παραγόμενων αρχιτεκτονικών που βασίζονται στο πρότυπο των Microservices, κρίθηκε απαραίτητη η εισαγωγή νέων, πιο εξειδικευμένων κριτηρίων, τα οποία αντανακλούν τις ιδιαιτερότητες και τις απαιτήσεις αυτής της αρχιτεκτονικής προσέγγισης. Τα κριτήρια αυτά στοχεύουν στην πληρέστερη και πιο ουσιαστική αποτίμηση της ποιότητας και της ορθότητας των προτεινόμενων λύσεων. Συγκεκριμένα, χρησιμοποιήθηκαν τα εξής:

1. **Ευθυγράμμιση με τις Λειτουργικές Απαιτήσεις & Κατανομή Ευθυνών**
   Εξετάζεται κατά πόσο κάθε μικροϋπηρεσία αντιστοιχεί σε έναν σαφώς οριοθετημένο επιχειρησιακό τομέα (bounded context) και αναλαμβάνει ένα συγκεκριμένο και συνεκτικό σύνολο λειτουργιών. Το σύνολο των λειτουργιών όλων των μικροϋπηρεσιών θα πρέπει να καλύπτει πλήρως τις λειτουργικές απαιτήσεις του συστήματος.

2. **Χαμηλή εξάρτηση & Ανεξαρτησία Ανάπτυξης**
   Αξιολογείται η χαλαρή σύζευξη (loose coupling) μεταξύ μικροϋπηρεσιών και η δυνατότητα αυτόνομης ανάπτυξης, αναβάθμισης και διάθεσής τους, χωρίς να επηρεάζονται άλλες υπηρεσίες.

3. **Συνοχή**
   Εξετάζεται η εσωτερική συνοχή των μικροϋπηρεσιών. Ιδανικά, κάθε μικροϋπηρεσία θα πρέπει να επιτελεί λειτουργίες που σχετίζονται στενά μεταξύ τους, ενώ η υλοποίηση ενός use case ενδέχεται να απαιτεί τη συνεργασία περισσότερων της μίας μικροϋπηρεσιών.

4. **Διαχείριση Δεδομένων**
   Αξιολογείται αν κάθε μικροϋπηρεσία διαχειρίζεται αυτόνομα τα δικά της δεδομένα και αν αποφεύγεται η χρήση κοινών βάσεων δεδομένων μεταξύ υπηρεσιών.

5. **Συνέπεια Δεδομένων**
   Εξετάζεται η πρόβλεψη μηχανισμών για την επίτευξη τελικής συνέπειας (eventual consistency) των δεδομένων, όταν αυτό απαιτείται από τη συνεργασία μεταξύ μικροϋπηρεσιών.

6. **Επικοινωνία & Έλεγχος Ροής**
   Ελέγχεται αν ο συντονισμός μεταξύ υπηρεσιών πραγματοποιείται μέσω κατάλληλων μηχανισμών. Επίσης, αξιολογείται η χρήση εργαλείων όπως API Gateways ή μηχανισμών ανταλλαγής μηνυμάτων (π.χ. publish-subscribe) για τον έλεγχο της ροής.

7. **Μη Λειτουργικές Απαιτήσεις**
   Αξιολογείται αν η αρχιτεκτονική ικανοποιεί τις μη λειτουργικές απαιτήσεις που τέθηκαν για το συγκεκριμένο πρόβλημα, όπως επεκτασιμότητα, διαθεσιμότητα, ασφάλεια κ.ά.

Μέρος αυτού του πειράματος ήταν η εισαγωγή ποσοτικοποιημένων δεικτών μέτρησης (από προηγούμενες έρευνες) για την αξιολόγηση αρχιτεκτονικής τύπου Microservices, με σκοπό να διερευνηθεί κατά πόσο μπορούν να προσφέρουν μια αντικειμενική και αξιόπιστη διάσταση στην διαδικασία της αξιολόγησης. Αυτό έγινε διαλέγοντας δείκτες που έχουν κάποια νοηματική συσχέτιση με τα υποκειμενικά κριτήρια αξιολόγησης, και ο τρόπος που εξετάσαμε αν αυτοί οι δείκτες μπορούν να βοηθήσουν είναι παρατηρώντας εάν εν τέλη πραγματικά υπάρχει συσχέτιση μεταξύ των υποκειμενικών κριτηρίων αξιολόγησης (human evaluation score) και των υπολογίσιμων δεικτών (calculatable metric).

Επίσης, οι δείκτες που επιλέχθηκαν υπολογίστηκαν τόσο χειροκίνητα από ανθρώπινους αξιολογητές όσο και αυτόματα από τα ίδια τα LLMs, ώστε να διαπιστωθεί εάν τα μοντέλα παρουσιάζουν φαινόμενα παραπλάνησης ή "ψευδαισθήσεων" (hallucinations) κατά τον υπολογισμό των δεικτών αυτών.

Περιληπτικά και με βάση τα παραπάνω, το prompt προς τα LLMs περιήχε τα εξής:

- **Γενική Περιγραφή Αποστολής**: Μια συνοπτική παρουσίαση του έργου που ανατίθεται στο γλωσσικό μοντέλο.

- **Περιγραφή Εφαρμογής**: Αναλυτική περιγραφή του λογισμικού-στόχου για το οποίο πρόκειται να παραχθεί η αρχιτεκτονική (μέσω εγγράφου SRS ή λίστας λειτουργικών/μη λειτουργικών απαιτήσεων - FR/NFR).

- **Οδηγίες Σχεδίασης για Αρχιτεκτονική Microservices**: Σύνολο βέλτιστων πρακτικών και αρχιτεκτονικών αρχών για τον σχεδιασμό Microservices.

- **Απαιτήσεις Σχεδίασης PlantUML**: Συγκεκριμένες οδηγίες για τη δομή του διαγράμματος κλάσεων.

- **Κατηγοριοποίηση Λειτουργιών στο Διάγραμμα Κλάσεων PlantUML**: Οδηγίες για την ταξινόμηση κάθε λειτουργίας (operation) των κλάσεων σε μία από τις εξής κατηγορίες: επιχειρησιακή λογική (business logic), διαχείριση δεδομένων (data management), διατήρηση συνέπειας δεδομένων (data consistency) ή έλεγχος ροής (flow control). Η ταξινόμηση αυτή υποστηρίζει την αξιολόγηση της αρχιτεκτονικής ποιότητας.

- **Περιγραφή των δεικτών μέτρησης προς Υπολογισμό**: Επισκόπηση των δεικτών που πρόκειται να εξαχθούν αυτόματα από τα παραγόμενα διαγράμματα.

- **Αναμενόμενη Μορφή Εξόδου των δεικτών μέτρησης (JSON)**: Η απαιτούμενη μορφή JSON για την παρουσίαση των υπολογισμένων δεικτών.

Αφού ολοκληρώθηκε η παραγωγή των διαγραμμάτων από τα LLMs, ακολούθησε η αξιολόγησή τους βάσει των προαναφερθέντων κριτηρίων και έπειτα ο υπολογισμός των δεικτών για κάθε μία από τις παραγόμενες αρχιτεκτονικές. Τα βασικά ευρήματα σχετικά με την ποιότητα των παραγόμενων διαγραμμάτων συνοψίζονται ως εξής:

1. **Τα μεγαλύτερα μοντέλα επιδεικνύουν γενικά καλύτερη απόδοση**: Τα μεγαλύτερα εμπορικά LLMs υπερίσχυσαν των μικρότερων τοπικών μοντέλων, παράγοντας συνολικά πιο πλήρη και δομημένα διαγράμματα.

2. **Τα LLMs ανταποκρίνονται ικανοποιητικά στον σχεδιασμό αρχιτεκτονικών Microservices**: Τα περισσότερα μοντέλα και ιδίως τα μεγαλύτερα επέδειξαν ικανοποιητική κατανόηση των αρχών σχεδίασης βάσει Microservices. Τα παραγόμενα διαγράμματα, αν και συχνά απαιτούσαν περαιτέρω βελτιστοποίηση, αποτελούσαν ένα αξιόλογο σημείο εκκίνησης.

3. **Η τεχνική RAG συνεισφέρει θετικά στις περισσότερες περιπτώσεις**: Η αξιοποίηση της μεθοδολογίας RAG οδήγησε γενικά σε βελτιωμένα αποτελέσματα. Αν και ορισμένα μοντέλα δεν επωφελήθηκαν ιδιαίτερα, ή παρουσίασαν οριακή υποχώρηση στην ποιότητα, η πλειονότητα παρουσίασε αισθητή βελτίωση όταν τους παρασχέθηκε στοχευμένο και συνοπτικό πληροφοριακό υλικό, βασισμένο σε θεμελιώδεις αρχές της αρχιτεκτονικής Microservices.

4. **Η μορφή εισόδου αποδεικνύεται καθοριστική**: Τα μεγαλύτερα μοντέλα διαχειρίστηκαν με μεγαλύτερη αποτελεσματικότητα τα αναλυτικά έγγραφα SRS σε σύγκριση με τις απλουστευμένες λίστες απαιτήσεων (FR/NFR).

5. **Οι αντικειμενικοί δείκτες παρουσιάζουν υποσχόμενες δυνατότητες**: Παρόλο που οι δείκτες δεν χρησιμοποιήθηκαν άμεσα ως εργαλείο αξιολόγησης στην παρούσα μελέτη, διαπιστώθηκε ισχυρή συσχέτιση μεταξύ αυτών και των υποκειμενικών αξιολογήσεων από ειδικούς. Το γεγονός αυτό αναδεικνύει τη μελλοντική δυναμική υιοθέτησης

αυτοματοποιημένων αξιολογήσεων, με χρήση δεικτών προσαρμοσμένων στην αρχιτεκτονική.

6. **Οι υπολογισμένοι δείκτες μέτρησης που παράγονται από τα LLMs παρουσιάζουν μεταβλητή ακρίβεια**: Κατά την αυτόνομη προσπάθεια των μοντέλων να υπολογίσουν τους δείκτες, παρατηρήθηκε ασυνέπεια στα αποτελέσματα. Η εμφάνιση «παραισθήσεων» (hallucinations) ήταν συχνό φαινόμενο, επισημαίνοντας την ανάγκη για εξωτερική επικύρωση και προσεκτική ερμηνεία των παραγόμενων δεδομένων.

Η πλήρης περιγραφή του πειράματος, με αναλυτική παρουσίαση όλων των παραμέτρων και των αποτελεσμάτων, παρατίθεται στο κυρίως αγγλικό κείμενο που ακολουθεί την παρούσα ελληνική περίληψη.

Πείραμα MyCharts με 2 Διαδοχικά Prompts

Στο πλαίσιο του πειράματος «MyCharts με 2 Διαδοχικά Prompts», εξετάστηκε αν η χρήση επανατροφοδότησης (feedback loop) με τη μορφή ενός δεύτερου, ειδικά διαμορφωμένου prompt μπορεί να οδηγήσει στη βελτίωση της ποιότητας των διαγραμμάτων που παράγονται από Μεγάλα Γλωσσικά Μοντέλα (LLMs).

Αφετηρία αποτέλεσε η διαδικασία του βασικού πειράματος «MyCharts», κατά την οποία τα μοντέλα καλούνται να παραγάγουν αρχιτεκτονικά διαγράμματα με βάση ένα σύνολο απαιτήσεων για τη συγκεκριμένη εφαρμογή. Αφού ελήφθη η πρώτη απάντηση από το εκάστοτε μοντέλο, η παραγόμενη αρχιτεκτονική αξιολογήθηκε με βάση τα ίδια κριτήρια που χρησιμοποιήθηκαν στο βασικό πείραμα. Παράλληλα, υπολογίστηκαν και οι αντίστοιχοι δείκτες που ενσωματώθηκαν στο προηγούμενο πείραμα.

Το καινούργιο στοιχείο του παρόντος πειράματος ήταν η εισαγωγή ενός δεύτερου prompt, το οποίο σχεδιάστηκε ως δομημένο και τυποποιημένο feedback προς το LLM. Το prompt αυτό περιλάμβανε τόσο τις ποιοτικές αξιολογήσεις των ειδικών όσο και τους ποσοτικοποιημένους δείκτες της πρώτης απάντησης, με σκοπό να καθοδηγήσει το μοντέλο προς την παραγωγή μιας πιο ορθής, πληρέστερης και συνεπέστερης αρχιτεκτονικής πρότασης. Με άλλα λόγια, αξιοποιήθηκε η αρχική έξοδος του μοντέλου ως βάση και ζητήθηκε αναθεώρηση ή βελτίωση, με βάση αντικειμενικά και υποκειμενικά κριτήρια.

Η δεύτερη απάντηση των μοντέλων αξιολογήθηκε εκ νέου με την ίδια μεθοδολογία και συγκρίθηκε άμεσα με την αρχική έξοδο. Με τον τρόπο αυτό, διερευνήθηκε εμπειρικά η αποτελεσματικότητα της επανατροφοδότησης μέσω διαδοχικής αλληλεπίδρασης (iterative prompting) και αν αυτή μπορεί να συμβάλει ουσιαστικά στη βελτίωση της αρχιτεκτονικής σκέψης και παραγωγής των LLMs.

Τα αποτελέσματα του πειράματος έδειξαν ότι η χρήση ενός δεύτερου, δομημένου prompt μπορεί πράγματι να συμβάλει στη βελτίωση της ποιότητας των παραγόμενων αρχιτεκτονικών διαγραμμάτων από τα LLMs. Αν και πρόκειται για ένα μικρής κλίμακας πείραμα, τα ευρήματα καταδεικνύουν τη δυναμική της μεθόδου και ενισχύουν την ιδέα πως μια τυποποιημένη, βασισμένη σε αξιολόγηση επανατροφοδότηση μπορεί να βελτιώσει τη σχεδιαστική ικανότητα

των μοντέλων. Προς το παρόν, η διαδικασία απαιτεί χειροκίνητη αξιολόγηση και υπολογισμό των δεικτών μετά την πρώτη απόκριση, ωστόσο το πείραμα αφήνει ανοιχτό το ενδεχόμενο για την ανάπτυξη ενός ημιαυτοματοποιημένου μηχανισμού βελτίωσης, ο οποίος θα ενισχύει τη χρηστικότητα των LLMs στο πεδίο της αρχιτεκτονικής λογισμικού.

Η πλήρης περιγραφή του πειράματος, με αναλυτική παρουσίαση όλων των παραμέτρων και των αποτελεσμάτων, παρατίθεται στο κυρίως αγγλικό κείμενο που ακολουθεί την παρούσα ελληνική περίληψη.

Συμπεράσματα

Τα αποτελέσματα αυτής της μελέτης αναδεικνύουν τις υποσχόμενες δυνατότητες ενσωμάτωσης των Μεγάλων Γλωσσικών Μοντέλων (LLMs) στη φάση σχεδιασμού του κύκλου ζωής ανάπτυξης λογισμικού. Μέσω της εξέτασης των μορφών εισόδου, της ποιότητας της εξόδου, της πολυπλοκότητας της αρχιτεκτονικής, των μεθόδων αξιολόγησης, της απόδοσης των μοντέλων και των τεχνικών βελτίωσης, αυτή η έρευνα παρέχει πρακτικές γνώσεις για το πώς η τεχνητή νοημοσύνη μπορεί να υποστηρίξει το αρχιτεκτονικό σχεδιασμό. Τα ευρήματα αυτά μπορούν να βοηθήσουν στον εντοπισμό των σημείων όπου τα LLMs προσθέτουν πραγματική αξία, στην αναγνώριση των τρεχουσών περιορισμών και στην ανάπτυξη στρατηγικών για την αποτελεσματική ενσωμάτωση της τεχνητής νοημοσύνης σε ρεαλιστικές ροές εργασίας σχεδιασμού.

Αντανακλώντας πίσω στα ερευνητικά μας ερωτήματα, επιχειρούμε τώρα να τα απαντήσουμε βάσει των ευρημάτων αυτής της μελέτης.

1. Η αξιολόγησή μας σχετικά με τις μορφές εξόδου αποκάλυψε ότι όταν ζητείται από τα LLMs να δημιουργήσουν αρχιτεκτονικές σε γλώσσα PlantUML, παράγουν σταθερά υψηλότερης ποιότητας αρχιτεκτονικές αναπαραστάσεις σε σύγκριση με άλλες μορφές. Η έξοδος PlantUML προσφέρει μια ισορροπία μεταξύ δομής και αναγνωσιμότητας που φαίνεται ιδιαίτερα κατάλληλη για τα LLMs.

2. Όσον αφορά την αναπαράσταση της εισόδου, διαπιστώσαμε ότι τα δομημένα έγγραφα Προδιαγραφών Απαιτήσεων Λογισμικού (SRS) μπορούν να βελτιώσουν την απόδοση των LLMs σε σύγκριση με απλά έγγραφα απαιτήσεων σε απλό κείμενο για μεγαλύτερα μοντέλα, που μπορούν να επωφεληθούν από το εκτενές πλαίσιο και την πλούσια δομημένη πληροφορία στα έγγραφα SRS. Αυτό υποδηλώνει ότι η ποιότητα και η μορφή των δεδομένων εισόδου παίζουν κρίσιμο ρόλο στην ακρίβεια της εξόδου των αρχιτεκτονικών που παράγονται από AI. Απλούστερες λίστες λειτουργικών (FR) και μη λειτουργικών απαιτήσεων (NFR) μπορεί να είναι χρήσιμες για μικρότερα μοντέλα, αλλά συχνά στερούνται του βάθους που απαιτείται για πιο πολύπλοκα σχέδια.

3. Όταν αντιμετωπίζουν πολύπλοκες σχεδιαστικές προκλήσεις, ειδικά αρχιτεκτονικές μικροϋπηρεσιών (Microservices), τα LLMs εμφάνισαν ανάμεικτη απόδοση. Τα μεγαλύτερα, πιο ικανά μοντέλα γενικά είχαν μεγαλύτερη επιτυχία στην αποσύνθεση πολύπλοκων απαιτήσεων και στην εφαρμογή βασικών αρχών μικροϋπηρεσιών, ενώ τα

μικρότερα μοντέλα συχνά δυσκολεύονταν. Αυτό τονίζει την ανάγκη η επιλογή των LLMs να είναι ανάλογη της πολυπλοκότητας της αρχιτεκτονικής του προβλήματος.

4. Μια ακόμα πτυχή αυτής της μελέτης είναι η εισαγωγή εξειδικευμένων κριτηρίων αξιολόγησης και αντικειμενικών δεικτών ειδικά για τις μικροϋπηρεσίες. Αυτή είναι η πρώτη προσπάθεια κάλυψης κενών στις υπάρχουσες μεθόδους αξιολόγησης και ταυτόχρονα συσχέτισης των υποκειμενικών ανθρώπινων αξιολογήσεων με ποσοτικοποιημένους δείκτες, ανοίγοντας το δρόμο για πιο τυποποιημένη αξιολόγηση της απόδοσης των LLM σε μελλοντικές εργασίες.

5. Παρατηρήσαμε επίσης ότι η Τεχνική Ενισχυμένης Ανάκτησης Πληροφοριών (Retrieval-Augmented Generation, RAG) βελτιώνει την ποιότητα του σχεδιασμού, ιδιαίτερα σε μεγαλύτερα μοντέλα που μπορούν να διαχειριστούν το αυξημένο πλαίσιο χωρίς να παραπλανηθούν ή να συγχυστούν. Το υλικό RAG φαίνεται να έχει μεγάλο αντίκτυπο στην ενίσχυση και όχι στην παραπλάνηση των LLMs τα αποτελέσματά μας δείχνουν ότι τα περιεκτικά και ακριβή υλικά RAG με πρακτικές οδηγίες λειτουργούν καλύτερα.

6. Παρά αυτές τις προόδους, η μελέτη αποκάλυψε και κάποιους περιορισμούς. Τα LLMs συχνά εμφανίζουν λάθη (hallucination) όταν τους ζητείται να υπολογίσουν αντικειμενικούς δείκτες στα διαγράμματα που οι ίδιοι δημιούργησαν. Αυτό υπογραμμίζει τη σημασία της εξωτερικής αντικειμενικής επικύρωσης και τους κινδύνους της αποκλειστικής εμπιστοσύνης σε αξιολογήσεις που παράγονται μόνο από AI.

7. Τέλος, το πείραμά μας με επαναληπτικές ερωτήσεις (iterative prompting) έδειξε θετικά αποτελέσματα, υποδηλώνοντας ότι μια δομημένη δεύτερη ερώτηση, που βασίζεται σε ανατροφοδότηση αξιολόγησης, μπορεί να βελτιώσει και να εξελίξει την ποιότητα του παραγόμενου διαγράμματος. Παρότι δοκιμάστηκε σε περιορισμένη κλίμακα, αυτό το εύρημα ανοίγει δυνατότητες για ημι-αυτοματοποιημένες ροές εργασίας βελτίωσης που συνδυάζουν την ανθρώπινη εποπτεία με τον AI-παραγόμενο σχεδιασμό.

Μελλοντική Έρευνα

Ενώ αυτή η μελέτη παρέχει ελπιδοφόρα ευρήματα σχετικά με τη χρήση των LLMs για τη δημιουργία αρχιτεκτονικής λογισμικού, παραμένουν αρκετοί τομείς προς διερεύνηση στο μέλλον. Μία σημαντική κατεύθυνση είναι η κλιμάκωση των πειραμάτων, τόσο ως προς τον αριθμό των περιπτώσεων χρήσης όσο και την ποικιλία των εξεταζόμενων αρχιτεκτονικών προτύπων. Αυτό θα επέτρεπε ευρύτερες γενικεύσεις και θα μπορούσε να αποκαλύψει αν οι τάσεις που παρατηρήθηκαν εδώ ισχύουν σε διαφορετικούς τομείς, μεγέθη έργων και εφαρμογές συγκεκριμένων βιομηχανιών.

Ένας ακόμα υποσχόμενος τομέας είναι η αυτοματοποίηση της διαδικασίας αξιολόγησης και βελτίωσης. Προς το παρόν, η μέθοδος με δύο εντολές (two-prompt method) που εισήχθη σε αυτήν την έρευνα βασίζεται σε χειροκίνητα υπολογιζόμενους δείκτες και ανθρώπινες αξιολογήσεις για την καθοδήγηση της βελτίωσης. Το λογικό επόμενο βήμα θα ήταν η ενσωμάτωση αυτών των διαδικασιών σε μια ημι-αυτοματοποιημένη ή πλήρως

αυτοματοποιημένη ροή εργασίας, που χρησιμοποιεί τυποποιημένους, επικυρωμένους ποσοτικοποιημένους δείκτες για την αξιολόγηση των αρχικών αποτελεσμάτων και τη δημιουργία αποτελεσματικών επακόλουθων ερωτημάτων (prompts). Αυτό θα μπορούσε να απλοποιήσει σημαντικά τη χρήση των LLMs στον αρχιτεκτονικό σχεδιασμό και να τα καταστήσει πιο πρακτικά για ενσωμάτωση σε ρεαλιστικές ροές ανάπτυξης.

Επιπλέον, υπάρχει σημαντικό περιθώριο βελτίωσης στην αντιμετώπιση των φαινομένων hallucination, ειδικά σε περιπτώσεις όπου ζητείται από τα LLMs να υπολογίσουν ή να επιχειρηματολογήσουν για συγκεκριμένες αρχιτεκτονικές με την χρήση ποσοτικοποιημένων διεκτών. Μελλοντικές μελέτες θα μπορούσαν να εξετάσουν τεχνικές όπως η ενσωμάτωση εξωτερικής γνώσης ή η λεπτομερής εκπαίδευση (fine-tuning) μοντέλων για τη μείωση των ανακρίβειών. Καθώς τα LLMs εξελίσσονται, η κατανόηση του πώς να στηρίζουμε αξιόπιστα τις εξόδους τους σε πραγματικά και συμφραζόμενα τεκμηριωμένα επιχειρήματα θα είναι κρίσιμη για την επιτυχή εφαρμογή τους στη τεχνολογία λογισμικού.

1. Introduction

The software development lifecycle is a complex, multistage process that spans from gathering user requirements to building and maintaining fully functional complex software systems. It involves several key phases, including requirement analysis, architecture and design, development, testing, deployment and maintenance. The system architecture phase of the software development lifecycle serves as the crucial bridge between abstract requirements and concrete implementation. During this stage, software engineers define the high-level structures and relationships that will form the backbone of the entire system. By creating model designs, often by utilizing Unified Modeling Language (UML), software engineers establish a blueprint that communicates the system's organization to all development teams. The system architecture determines not only how functional requirements will be met but also how non-functional requirements like scalability, security and performance will be addressed. A well-designed software architecture helps reduce development errors, sets clear technical standards and provides a stable foundation for future development. However, conceiving an architecture and creating the corresponding diagrams is often a time-consuming and error-prone task, with a high risk of misinterpreting widely available architectural principles. Such issues can lead to inefficiencies that affect the entire software development lifecycle.

The rise of artificial intelligence (AI), particularly through advanced large language models (LLMs) built on deep learning, has introduced exciting new ways to automate tasks that software engineers once had to handle manually. These AI systems, which learn from massive amounts of data, have gotten impressively good at working with human language, both understanding it and creating it. This skillset has already proven useful for coding-related tasks like generating code snippets and fixing bugs, which brings up the challenges of exploring AI tools' potential for something even more ambitious: automating the crucial creative architecture design phase and creating the corresponding UML diagrams automatically. Being able to use AI to turn written software requirements into well-structured architectural diagrams that comply with selected architectural styles would tackle one of the toughest challenges in software engineering. This approach could transform how teams handle what many consider the single creative "make-or-break" phase of building software.

This study aims to investigate the capability of large language models (LLMs) to generate software architectures, with particular focus on Microservices. We're building upon previous research by examining various inputs, ranging from plaint-text functional and non-functional requirements to complete standards-based Software Requirements Specification documents and by analyzing how model selection and Retrieval-Augmented Generation techniques influence the quality of the resulting architectures. Our primary concern is whether these AI-generated designs adequately satisfy all requirements while correctly implementing the specified architectural principles. We introduce a set of evaluation principles for assessment by human experts and we propose a set of metrics for objective evaluation, based on established metrics. Additionally, we're investigating whether smaller, locally deployed LLMs can serve as viable alternatives to commercial solutions for architectural design tasks.

In the subsequent sections, we will describe our research methodology and experimental design, present an analysis of our findings and conclude by discussing the broader implications of our results and proposing promising directions for future work in the field of AI-assisted software architecture design. Our discussion will particularly focus on how these advancements might transform the architectural phase of the software development lifecycle and the potential impacts on architectural quality, development efficiency and the evolving role of software architects working alongside AI systems.

1.1 Evolution of AI

Artificial Intelligence (AI) is reshaping the world by introducing groundbreaking efficiency in solving problems. At its essence, AI focuses on developing systems capable of performing tasks that normally require human intelligence such as reasoning, decision-making and understanding natural language. By utilizing algorithms and large volumes of data, AI systems can adapt to changing environments and improve their performance over time. Due to this constant evolution, AI is not only reshaping how we approach complex tasks now but also redefines the future of technology and innovation.

Artificial Intelligence (AI) is implemented using advanced computational methods such as machine learning and deep learning. Machine learning equips systems with the ability to recognize patterns and make predictions by processing extensive datasets [1]. Deep learning, a more advanced subset of machine learning, utilizes neural networks to address challenges like image classification and speech recognition [2]. These methods have significantly enhanced how machines interpret complex, high-dimensional data, laying the groundwork for many AI-driven solutions. A key area built on these foundations is natural language processing (NLP), which focuses on making machines able to understand, produce and engage with human language. NLP is necessary for technologies such as voice assistants, automated translation services and sentiment detection tools to work [3].

Artificial Intelligence (AI) is transforming a wide array of industries and becoming a key part of everyday life. In healthcare, AI-powered diagnostic systems enhance both accuracy and efficiency, while in transportation, autonomous vehicles rely on real-time data and predictive modeling to navigate safely and effectively. Tools like TensorFlow, PyTorch and OpenAI's models equip researchers and developers with the means to build complex systems with greater speed and flexibility [4]. These platforms support rapid development, scalability and seamless integration into existing infrastructures. However, as AI continues to evolve, it also brings forth critical ethical challenges related to bias, privacy and the transparency of automated decision-making, highlighting the urgent need for thoughtful regulation and responsible innovation [5].

1.2 AI in Software Engineering

The ability of AI tools to replicate aspects of human creativity has made them increasingly appealing in the field of software engineering [6]. By automating tasks such as code

generation, refactoring and debugging, AI has the potential to significantly enhance productivity and contribute to the development of higher-quality software. Large language models (LLMs) can consistently generate code, identify and fix errors, allowing them to handle many programming tasks either partially or, in some cases, completely. In addition to supporting development activities, AI can optimize team workflows by analyzing operational patterns and recommending process improvements. However, the integration of these capabilities into the software development lifecycle is challenging. It requires a deep understanding of system requirements, the specific development environment and the inherent limitations of current AI technologies [6].

While AI holds great promise for supporting every stage of the software development lifecycle, achieving this potential depends on overcoming several challenges. Utilizing large language models (LLMs) for tasks such as generating architectural designs, automating test procedures or forecasting development bottlenecks demands careful attention to accuracy, completeness and adherence to established design principles. These AI capabilities are not universally applicable in a plug-and-play manner, instead, they require planning, evaluation and customization to align with specific project requirements. Additionally, factors such as the quality and relevance of training data, domain-specific limitations and broader ethical implications must be considered to ensure reliable and responsible integration.

Ultimately, AI serves to augment, rather than replace, human expertise in software engineering. By creating systems that can learn, adapt and evolve, AI drives innovation and helps development teams handle the growing complexity of modern software applications. However, integrating AI into the software development lifecycle involves more than simply adopting tools, it requires an understanding of the associated trade-offs, limitations and the ethical and practical responsibilities that come with deploying AI [7], [8].

This literature review presents an overview of contemporary research examining how AI supports various aspects of software engineering. The studies are categorized by domain, highlighting key applications, potential benefits and existing challenges associated with AI integration.

1.2.1 Automated Code Generation

The emergence of large language models (LLMs) and Generative AI has introduced new possibilities in automated code generation, changing how software is developed and maintained. By incorporating machine learning, deep learning and natural language processing techniques, these systems can assist developers in tasks ranging from code completion to debugging.

Generative AI, powered by machine learning (ML) and deep learning (DL) algorithms, has played a crucial role in automating key elements of the code generation process. As noted by Tembhekar, these technologies allow the automation of tasks that were once the domain of human developers, significantly boosting efficiency within DevOps workflows [9]. The incorporation of natural language processing (NLP) techniques further enhances this capability,

allowing AI systems to interpret and generate code with increasing detail, thereby optimizing development processes and streamlining operations.

AI-driven code completion has become a standard tool used in software development, playing a vital role in boosting developer efficiency. Gao traces the transition from traditional statistical approaches to modern neural models, emphasizing in their effectiveness in streamlining the coding process [10]. However, the growing reliance on generative AI also raises critical ethical concerns. Atemkeng et al. [11] emphasize the importance of using these tools responsibly, calling for protective measures to ensure that generated code meets both functional and security standards. As AI continues to influence software engineering practices, these issues highlight the pressing need for thoughtful governance and accountability.

In summary, the incorporation of AI into code generation marks a major milestone in the evolution of software engineering. Current research in this area remains focused on striking an effective balance between the efficiencies of automation and the ethical challenges it presents, aiming to ensure that these powerful technologies are applied thoughtfully and responsibly.

1.2.2 Automated Testing

Automated testing using AI technologies is a promising approach to enhancing software quality and reducing testing durations. Mulla highlights that AI-driven testing allows the execution of tests with each software update, supporting continuous integration and delivery [12]. This capability is especially critical in modern development environments, where rapid iterations are the norm. Additionally, Job highlights that the growing complexity of software systems requires automated testing to maintain quality standards within shorter evaluation timelines [13].

The use of machine learning (ML) techniques in automated testing has been widely explored. For example, Gautam et al. offer an extensive review of how ML algorithms can be utilized to automate error detection, thereby improving the reliability of software systems [14].

On the contrary, the adoption of AI in automated testing comes with several challenges. Marijan points out key challenges in testing machine learning systems, notably the need for specialized testing frameworks that address the distinct characteristics of AI models [15]. The paper outlines a research focused on improving testing practices for machine learning applications, with an emphasis on developing robust and reliable testing methodologies. Similarly, Latika Kharb's work discusses the critical need for trust and transparency in automated testing within machine learning systems [16].

Overall, the literature suggests that AI has the potential to significantly transform automated testing in software engineering by improving efficiency, accuracy and reliability. However, the challenges associated with testing AI systems underscore the need for continued research and development to create methodologies that can address the distinctive requirements of this evolving field.

1.2.3 Understanding Software Requirements using Natural Language Processing

Natural Language Processing (NLP) plays a crucial role in automating and enhancing the traditionally manual processes involved in software requirements specification, elicitation and analysis. NLP techniques allow machines to process and understand human language, bridging the gap between human communication and computational systems. This section delves into the application of NLP in software requirements engineering and design, examining its impact on the quality, efficiency and automation of these processes.

A literature review by Calle and Zapata introduced the QUARE model (Question Answering for Requirements Elicitation), a novel NLP-based framework designed to improve the requirements engineering (RE) process. QUARE uses question-answering capabilities to assist software analysts in extracting relevant information from requirements documents, regardless of the writing style or structure [17]. This model emphasizes the potential of NLP for streamlining requirements elicitation by automating the extraction of information and providing software analysts with the tools needed to handle the complex and often ambiguous nature of natural language requirements. The application of NLP in requirements engineering, often referred to as NLP4RE, has gained traction as a valuable method for improving the accuracy and efficiency of requirements analysis, reducing human error and allowing faster decision-making [18], [19].

The integration of machine learning (ML) techniques alongside NLP further enhances the capabilities of these tools in the requirements engineering domain. Machine learning algorithms can be trained on large datasets of requirements documents, enabling them to recognize patterns and extract important insights. This synergy between NLP and ML has the potential to automate manual tasks such as requirements classification, conflict detection and validation, significantly improving the efficiency of software engineers. For example, NLP-based systems can automatically classify requirements into functional and non-functional categories, helping teams organize and prioritize the scope of a project [18]. This not only accelerates the elicitation process but also ensures that requirements are captured and categorized with greater precision.

An important application of NLP in software engineering is converting natural language requirements into formal representations that can be directly used in the development process. Semantic parsing techniques, which involve mapping natural language expressions to formal models, play a crucial role in this transformation. By applying semantic parsing, requirements can be structured in a way that allows for more precise interpretation and implementation by developers and other stakeholders [20]. This formalization process is critical for ensuring that requirements are not only understandable but also executable, making it easier for teams to transition from high-level specifications to detailed design and development.

However, despite the promising potential of NLP in software engineering, several challenges remain. A substantial portion of software requirements is still expressed in natural language, which often leads to ambiguity, inconsistency and misinterpretation. Research indicates that approximately 95% of software requirements are written in natural language, highlighting the challenges in ensuring clarity and accuracy in these specifications [21]. The

complexity of human language presents a significant obstacle for NLP systems, which must be trained to handle such challenges. As a result, misunderstandings and miscommunications between stakeholders are common, underscoring the need for advanced NLP tools that can better understand and validate these requirements.

Additionally, NLP-based systems must account for the diversity of requirements documentation styles and formats. Requirements documents may vary significantly in structure, terminology and level of detail, which can complicate the task of automatic analysis. Addressing these issues requires the development of more advanced NLP models that can adapt to diverse documentation practices and accurately extract relevant information. This ongoing research in NLP for requirements engineering is important for overcoming these challenges and improving the overall quality and reliability of requirements documentation.

In conclusion, the integration of AI, particularly NLP and ML, into software requirements engineering holds promise for improving the accuracy, clarity and efficiency of the requirements specification and design processes. While progress has been made, the field still faces challenges, particularly in handling the ambiguity and variability inherent in natural language requirements. As AI continues to evolve, it is likely that these technologies will reshape the landscape of software engineering, driving improvements in both the quality and speed of software development.

1.3 Challenges of AI in Software Engineering

While the integration of Artificial Intelligence (AI) into software engineering promises substantial gains in productivity, automation and quality, it also introduces complex challenges that must be carefully addressed. These challenges impact all phases of the software development lifecycle, from requirements engineering to development and testing.

One of the primary challenges is the verification and validation of AI-based systems. Traditional software verification techniques, which are based on deterministic behaviour, struggle to accommodate the probabilistic and data-driven nature of AI algorithms. These systems often operate on large and sometimes opaque datasets, which not only makes behaviour harder to predict but also introduces risks such as hidden biases, data drift and unexpected outcomes [22]. The dynamic and evolving nature of machine learning (ML) models further complicates quality assurance efforts, as conventional static testing frameworks are not capable for assessing systems that learn and change over time [23].

Another significant concern is the assurance of data quality and algorithmic transparency. AI models are only as reliable as the data they are trained on, yet real-world datasets often contain noise, biases, or imbalances. These issues can propagate through the AI pipeline, leading to inaccurate or discriminatory results. Moreover, many modern AI techniques, particularly deep learning, are "black box" in nature, lacking the explainability required for high-stakes software systems where decision accountability is crucial [24]. The demand for explainable AI (XAI)

continues to grow as stakeholders seek not only high performance but also clarity on why an AI system made a particular decision.

In software engineering, AI hold potential for automating various design-related tasks, including the generation of architectural diagrams and early detection of design flaws. However, realizing this potential remains a work in progress, as several critical limitations persist:

1. **Ambiguity and Complexity in Natural Language Requirements**
   Most software requirements are written in unstructured natural language, which is prone to ambiguity, vagueness and inconsistency. This lack of precision hinders AI systems' ability to extract actionable design elements automatically. While tools exist to assist in generating UML class diagrams, they often require significant human intervention to resolve ambiguities and fill in missing semantics such as class responsibilities, relationships and behavioural patterns [25].
2. **Difficulty in Inferring Relationships and Structural Hierarchies**
   A key challenge in class diagram and architecture generation is deducing relationships between components, such as inheritance, aggregation and dependency. Requirements document rarely spell out these relationships explicitly. Instead, they rely on contextual cues that require abstract reasoning and domain-specific interpretation capabilities that LLMs are not inherently optimized for [26].
3. **Limited Exposure to Formal Software Modelling Constructs**
   Although LLMs are trained on massive datasets that include code and documentation, they often lack sufficient representation of formal software design models, such as UML, design patterns and architecture blueprints. As a result, LLMs may fail to adhere to the syntactic and semantic conventions required for accurate modelling, leading to inconsistent or technically incorrect diagram outputs.
4. **Insufficient Domain Context and Pattern Awareness**
   Effective architectural decisions often depend on domain-specific knowledge and the application of established design patterns. LLMs typically do not possess the contextual grounding necessary to apply such patterns correctly. Research by Vaidhyanathan et al. [27] suggests that while LLMs can assist in documenting and iterating on design decisions, they fall short of autonomously generating coherent and context-aware architectural solutions. Rather than replacing human architects, these models are better suited to acting as co-pilots, augmenting human decision-making with automated suggestions and refinement support.

Additionally, ethical considerations such as fairness, accountability and transparency must be addressed in AI-enabled software engineering tools. As these tools increasingly influence critical system design and operational decisions, ensuring that their outputs align with ethical standards and societal expectations becomes necessary.

Motivated by these challenges, our research aims to explore the evolving capabilities of AI in supporting software architecture and design. In the following sections, we outline our experimental methodology and present results highlighting the strengths and limitations of LLMs in AI-assisted software engineering.

2. AI-Assisted Software Architecture

In this chapter, we examine the current landscape of AI-assisted software architecture, detail architectural patterns and explain how our approach stands apart from and builds upon previous research.

2.1 Related Work

The automatic generation of Unified Modeling Language (UML) diagrams from textual requirements has recently gained significant traction, largely due to advances in large language models (LLMs). This emerging area of research explores how cutting-edge AI technologies can help address long-standing challenges in software architecture, particularly bridging the gap between natural language requirements and formally designed artifacts.

This study builds upon the work of Tsilimigkounakis [28], who investigated the use of LLMs to automatically generate class diagrams from textual descriptions of a simple application, focusing on straightforward architectural patterns such as Client-Server, Three-Tier and Model-View-Controller. Building on that foundation, our research further investigates the potential of LLMs in supporting the design phase of the software development lifecycle, while preserving the core experimental methodology and research perspective established in the previous study.

Other methods for generating UML diagrams primarily utilized NLP techniques supported by rule-based systems or domain-specific ontologies. While these approaches showcased the potential for automation, they were limited by strict input requirements and frequent reliance on human oversight. Ambiguities in natural language posed significant challenges, often resulting in outputs that were fragmented or failed to fully capture the intended system design.

A promising approach is proposed by Eisenreich, Speth and Wagner, who outline a six-step framework designed to bridge the gap between textual requirements and software architecture generation [29]. The process starts with the automated generation of a domain model and use-case scenarios derived from natural language specifications. These initial outputs undergo manual refinement to improve their accuracy and contextual relevance. Using the refined domain model, scenarios and non-functional requirements, the system generates multiple architectural candidates along with associated design decisions. These candidates are then automatically evaluated and compared. The process concludes with manual refinement of the top candidates and the final selection of the most suitable architecture for implementation. In their exploratory study, the authors experimented with large language models (LLMs), including LLaMA2 70-B and GPT-3.5, to generate domain models directly from textual requirements. By prompting the models with requirement documents and instructing them to produce PlantUML domain diagrams, they observed that although the models effectively identified key domain concepts, they often misinterpreted the intended task. Rather than modeling the domain

contextually, the LLMs tended to generate representations of the system itself, revealing a mismatch between the prompt's objective and the model's output.

In a different line of research, Yang and Sahraoui proposed an AI-driven approach to mitigate the challenges posed by ambiguity in natural language during UML class diagram generation [30]. Their technique utilizes a machine learning-based binary classifier to determine whether each sentence in each input describes a class or a relationship. The methodology involves parsing English text into individual sentences, converting each into a corresponding UML diagram fragment and then assembling these fragments into a complete, coherent diagram. To support their model, the researchers curated a dataset consisting of UML diagrams and their associated English descriptions, establishing a direct mapping between natural language and UML components. This dataset, although relatively small, was created through crowdsourcing and proved adequate for training and evaluation purposes. While the approach showcased promising innovation, the resulting diagrams exhibited limited accuracy. The authors attributed this limitation to the shortcomings of the NLP tools used, suggesting that the adoption of more advanced NLP technologies could substantially improve both the precision and robustness of the outcomes, highlighting considerable potential for future advancement in this domain.

Building upon the capabilities of large language models, Iyad and Areen introduced *Vlissingen*, a tool that uses GPT-3.5's natural language processing (NLP) capabilities to automate the extraction of UML class diagram elements [31]. This approach marks a significant advancement in the field by employing few-shot learning to enhance GPT-3.5's effectiveness in diagram generation. By fine-tuning the model on a dataset of 50 varied case studies, each consisting of a prompt paired with an ideal output, *ClassDiagGen* achieved impressive results, reporting a precision of 98.6% and recall of 93.3%, thereby substantially outperforming previous techniques. The tool features an integrated pipeline that combines textual analysis, automatic PlantUML code generation and diagram rendering into a unified workflow. These results underscore the practical viability of large language models in automating and optimizing early-stage software design.

The literature reveals a clear progression toward more advanced systems capable of handling natural language ambiguities inherent in software requirements. These developments demonstrate that LLMs offer considerable promise for automating the transition from textual requirements to formal design artifacts. While challenges remain, the trajectory of research suggests that automated UML generation is becoming increasingly viable for practical application in software development processes. Our work builds upon these foundations to further explore the capabilities and limitations of LLMs in supporting software architecture design.

2.2 Architectural Patterns

This section examines the most common software architectural patterns, describing their structure and main components. It is designed to serve as a clear and practical guide for understanding and applying these patterns in software design.

2.2.1 Client-Server Architecture

The client-server architecture is a foundational and widely adopted design pattern in computer networking where system functionality is divided between service providers (servers) and service requesters (clients). This model is characterized by a clear separation of concerns: servers host, manage and deliver resources or services, while clients initiate communication to request and consume these services.

In a typical client-server system, servers are dedicated machines or processes that offer specific services, such as data storage, computation, or application functionality. Clients, which are often user-facing applications, interact with servers over a network to perform operations like querying a database, submitting a request for computation, or retrieving files. The interaction between clients and servers typically follows a request-response communication model, where the client sends a request and the server responds accordingly [32].

The main components of the client-server architecture include:

- **Servers**, which are specialized systems responsible for providing particular services. Examples include web servers (serving web content), file servers (managing file operations) and database servers (handling data storage and queries).

- **Clients**, which are applications or devices that initiate requests to the servers. A client may be a web browser accessing a website, a mobile application querying a cloud service, or a desktop application requesting data from a remote database.

- **Network infrastructure**, which allows communication between clients and servers, typically via standardized protocols such as TCP/IP, HTTP, or WebSocket over local networks or the Internet.

Client-server architectures allow multiple clients to interact concurrently with centralized servers. The reliability and manageability of this pattern have led to its widespread use across a variety of domains, including web applications, enterprise systems and cloud computing platforms. However, while the client-server model supports efficient resource sharing and centralized control, it also introduces potential challenges such as server bottlenecks, single points of failure and the need for robust security measures to protect data and services.

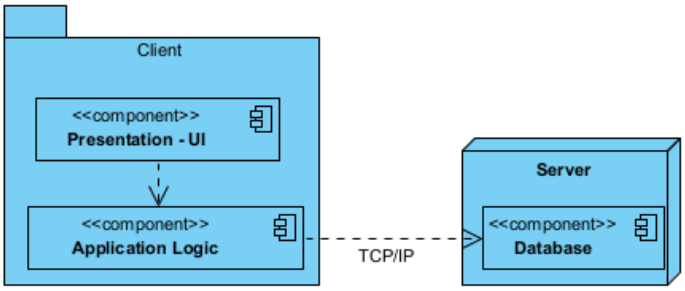


Figure 2.1: Client-Server Model

By clearly separating responsibilities between service providers and consumers, the client-server architecture remains a cornerstone of modern software and system design, producing scalable and maintainable applications.

2.2.2 Three-Tier Architecture

The Three-Tier Architecture is a well-established software design model that organizes applications into three distinct, logically separated layers: the Presentation Layer, the Business Logic Layer and the Data Layer. Each layer is responsible for specific aspects of the application's functionality, promoting separation of concerns, scalability and maintainability [33].

- **Presentation Layer**: The Presentation Layer is responsible for managing all interactions with the user. It includes user interfaces such as web pages, forms, or mobile screens and is designed to display data to users and capture their inputs. This layer is concerned solely with the presentation of information and the forwarding of user-initiated actions to the Business Layer. By isolating user interface components from business logic, the architecture supports easier updates and enhancements to the user experience without affecting the underlying application functionality.

- **Business Logic Layer (Application Layer)**: The Business Logic Layer contains the core functionality and business rules of the application. It acts as an intermediary between the Presentation and Data Layers, processing user requests, enforcing business policies and coordinating the flow of data. This layer ensures that input received from the user interface is validated, business rules are applied consistently and the appropriate data operations are triggered. By isolating the business logic, this layer supports reusability and allows the application to adapt more easily to changes in business requirements.

- **Data Layer (Persistence Layer)**: The Data Layer is responsible for managing the application's data storage and retrieval operations. It includes database management systems, data access objects and query processing components. This layer handles the complexities of database interactions from the upper layers by providing standardized methods to create, read, update, or delete data. Isolating the data access mechanisms

allows changes to the underlying data source (e.g., switching from one database system to another) without affecting the business logic or presentation.

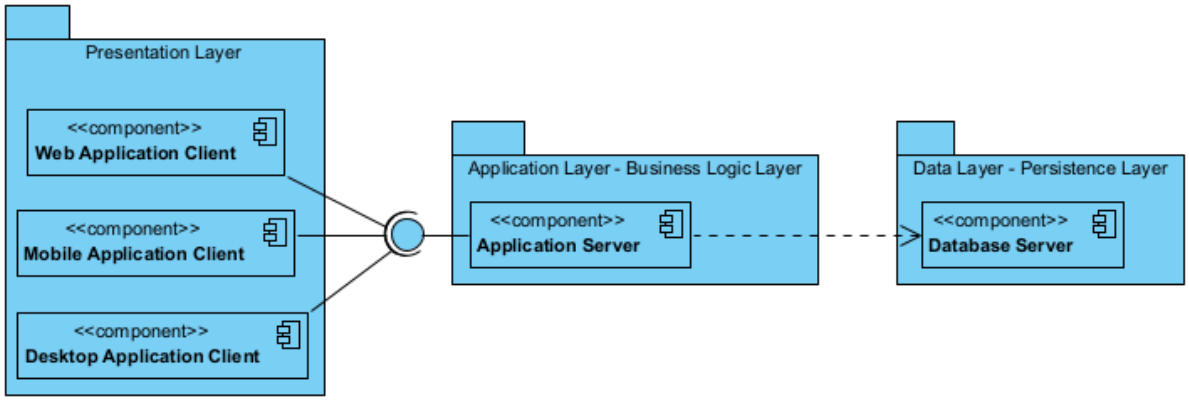


Figure 2.2: Three-Tier Architecture Model

The Three-Tier Architecture pattern enhances system scalability by allowing each layer to be scaled independently based on demand (e.g., using load balancers, replicating databases, or expanding application servers). It also improves maintainability, as developers can update or replace components within a single layer without impacting others, supporting agile development practices and long-term system evolution.

2.2.3 Model-View-Controller (MVC) Architecture

The Model-View-Controller (MVC) architectural pattern is a fundamental design paradigm for developing interactive software applications. It promotes a clear separation of concerns among the application's data, user interface and control logic, enhancing modularity and maintainability [34].

- **Model**: The Model contains the core functionality of the application, representing its data, business rules and operations. It manages the state of the application independently of the user interface and directly reflects the underlying domain or business logic. Changes to the model are typically propagated to interested views via notification mechanisms (e.g., observer pattern). Data persistence and retrieval between the database and the model layer are managed through *Data Access Objects* (DAOs) and *Data Transfer Objects* (DTOs), ensuring abstraction and decoupling from storage-specific concerns.

- **View**: The View is responsible for rendering the visual representation of the model's data. Each view accesses the model to display an up-to-date and specific subset of information to the user. It acts as a dynamic projection of the model's state, reflecting changes without altering the underlying data. Views often use filtering and formatting to present complex data structures in a user-friendly manner. In some implementations, the

view can subscribe to the model to receive automatic updates upon state changes, supporting reactive interfaces.

- **Controller**: The Controller serves as the intermediary between the user, the model and the view. It handles user inputs, processes them (possibly invoking business logic on the model) and determines the appropriate view for response. Controllers interpret user actions (e.g., mouse clicks, form submissions) and invoke corresponding changes on the model. After processing, they coordinate the selection and rendering of a view to reflect the new state. In web applications, controllers typically map HTTP requests to application functionality and decide how to respond, often using frameworks like Spring MVC or Struts in Java EE ecosystems.

By decoupling the components, MVC allows for independent development, testing and maintenance of the model, view and controller layers. It also supports multiple views of the same model, allows easier integration of new interfaces and facilitates parallel development by separating front-end and back-end concerns.

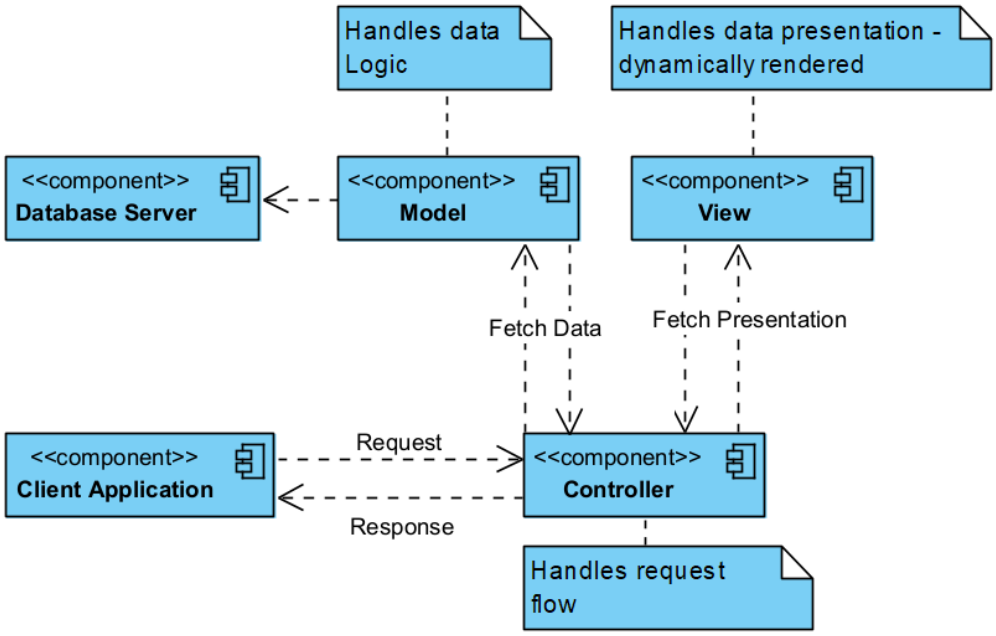


Figure 2.3: Model-View-Controller (MVC) Architecture Model

2.2.4 Microservices Architecture

The microservices architectural pattern represents a modern approach to software design characterized by the decomposition of applications into independent, autonomous services, each responsible for a specific business function. Unlike traditional monolithic systems, microservices implement vertical decomposition, allowing each service to be developed, deployed and scaled

independently. This architecture has gained significant adoption among organizations of all sizes, from industry leaders like Amazon, Netflix and Spotify to small and medium enterprises seeking greater flexibility and resilience in their systems.

Microservices offer numerous technical advantages compared to traditional architectures. Each service can be built using the most appropriate programming language and technology stack for its specific requirements, enabling targeted optimization. Services scale individually based on demand, allowing for efficient resource allocation. The decentralized, modular structure also enhances system maintainability and fault tolerance, as the failure of one service does not compromise the entire application which is a common vulnerability in monolithic systems.

While microservices share conceptual roots with Service-Oriented Architecture (SOA), they represent a distinct evolution. Where SOA services typically encompass complete business capabilities, microservices focus on smaller, highly specialized software components dedicated to single tasks. This granular approach overcomes many limitations of traditional SOA implementations, making microservices particularly suitable for modern cloud-based enterprise environments where flexibility, scalability and resilience are paramount considerations.

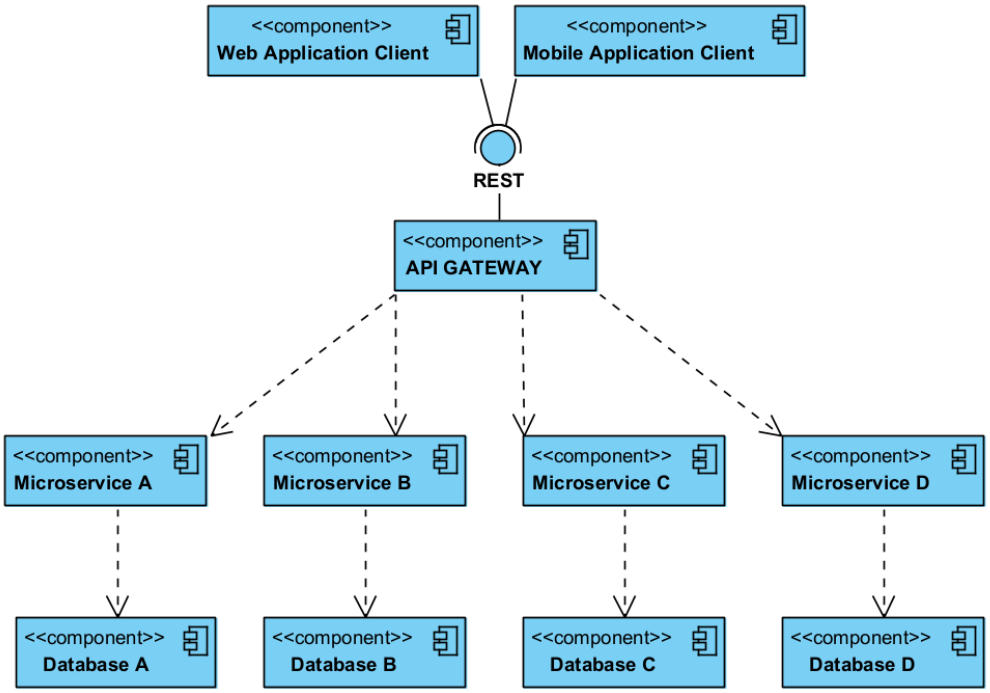


Figure 2.4: Microservices Architecture Model

2.3 Selected Architectural Patterns

This research builds on previous investigations into established architectural patterns such as Client-Server, Three-Tier and Model-View-Controller (MVC) while expanding our scope to include the Microservices architectural paradigm.

Our methodology follows a two-phase approach in terms of selected architectures. First, we will continue our established research trajectory by implementing a simple application using the simpler architectural patterns such as Client-Server, Three-Tier and MVC. Unlike previous work, this phase will incorporate well-structured requirements through a formal Software Requirements Specification (SRS) document, to evaluate if more detailed requirements help the model to produce better architectures. Following the evaluation process, we will progress to analysing AI capabilities in handling more complex architectural challenges by implementing a more complex application with more requirements within a Microservices Architecture framework.

Through this structured comparison, we aim to systematically evaluate AI's ability to generate and implement architectural solutions across varying levels of complexity, with particular emphasis on its competence to produce coherent designs for moderately complex systems within the Microservices paradigm.

2.4 Motivation of Our Approach

The evolution of AI technologies, particularly Large Language Models (LLMs), has created immense opportunities to transform software development practices. This research is motivated by the growing need to determine whether modern AI can effectively assist in the complex process of architectural design that has traditionally relied on human expertise and experience. As organizations increasingly seek to accelerate development cycles and address talent shortages, understanding AI's capabilities and limitations in this critical domain becomes essential for responsible integration into professional software engineering workflows.

The architectural design phase represents one of the most intellectually demanding aspects of software development, requiring engineers to translate abstract business requirements into concrete system structures that balance numerous technical and operational concerns. Current approaches to AI assistance in software development have primarily focused on implementation-level tasks, leaving a significant opportunity to explore how AI might augment architectural thinking. Our approach is motivated by the hypothesis that properly trained LLMs could potentially assist in architectural reasoning, pattern selection and design representation. Thus, providing valuable support to human architects while addressing common challenges such as consistency, documentation and alignment with requirements. By investigating this potential, we aim to establish a foundation for understanding where and how AI can meaningfully contribute to architectural processes.

Lastly, the rapid advancement of AI capabilities requires the development of alternative objective evaluation methodologies. Unlike code generation, where correctness can often be objectively verified through compilation or test execution, architectural quality contains more subjective dimensions such as modularity, scalability and appropriateness for business context. Our research is motivated by the need to propose evaluation frameworks and metrics that can meaningfully assess AI-generated architectural designs across these dimensions. By introducing and testing such metrics, we lay the groundwork for future evaluation tools to be implemented. Tools that will be able to benchmark advancements in AI's architectural capabilities as AI technologies continue to evolve. This motivation drives our investigation into the connection of AI and software architecture, seeking to define the current boundaries of possibility and the pathways toward more advanced AI assistance in architectural design.

2.5 Research Questions

We aim to explore AI's capabilities and limitations in software development assistance, focusing specifically on architectural design. Specifically, we investigate how AI can generate software architectures, what materials improve model training for this task and how to evaluate AI performance in architectural design.

1. **Which output format yields optimal architectural representations from Large Language Models?**
   This investigation aims to determine the most effective format for AI-generated architectural designs. We evaluate various representation methods including XMI/UML, images, PlantUML and Mermaid diagrams to identify which format allows LLMs to produce the highest quality and most accurate architectural designs.

2. **Does a structured Software Requirements Specification Document enhance AI's architectural design capabilities?**
   We explore whether AI produces better software designs when presented with SRS documentation versus simplified requirement lists. We examine how effectively AI interprets complex business logic, use cases and interconnected requirements when translating them into coherent architectural designs.

3. **How effectively can AI develop Microservices architectures for complex problem domains?**
   We test AI's capability to decompose complex requirements into appropriate microservice components while correctly implementing architectural principles such as service boundaries, relationships and inter-service communication patterns.

4. **What evaluation criteria best assess AI-generated Microservices architectures?**
   We address the limitations of existing evaluation frameworks when applied to Microservices architectures. We aim to develop specialized criteria that more accurately assess the unique aspects of microservice design that weren't captured in our previous evaluations of simpler architectural patterns.

5. **How can we utilize objective metrics for AI-generated architectures?**
   As LLM technology rapidly evolves, we need automated methods to objectively measure improvements in architectural design capabilities. We focus on creating architecture-specific metrics that provide consistent, quantifiable evaluation of AI performance.

6. **Do LLMs hallucinate when calculating well-defined metrics for the class diagrams they generated?**
   We examine whether LLMs can calculate objective well-defined metrics accurately from the class diagrams they generated. We measure hallucination and conceptual mistakes that might reveal gaps in the model's understanding.

7. **How does Retrieval-Augmented Generation (RAG) impact architectural design quality?**
   We explore if RAG techniques can enhance context-awareness and design accuracy, improving AI's architectural reasoning capabilities.

8. **How does iterative prompting affect the quality of AI-generated architectural designs?**
   We test whether multi-turn conversations, without explicit knowledge injection, progressively improve the quality of architectural designs. We analyze how iterative refinement through conversation affects the completeness, correctness and coherence of the resulting architecture.

These research questions collectively address some of the fundamental challenges of integrating AI into software architecture development workflows. By investigating input formats, output representations, architectural complexity boundaries, evaluation methodologies, model capabilities and enhancement techniques, this research aims to establish practical guidelines for utilizing AI in architectural design. The findings will help software engineering teams better understand where AI can provide genuine value in the architectural process, identify current limitations and implement strategies to optimize AI-assisted design outcomes for real-world software projects.

3. Approach

3.1 Deployment, setup and technologies used

Before diving into the details of our experimental setup and results, it's important to highlight a few constants that stayed the same throughout all experiments. These include the hardware used in our testing environment, the tools and technologies we relied on and the RAG (Retrieval-Augmented Generation) methods we applied. Outlining these shared elements helps provide a consistent foundation for understanding and comparing our findings.

3.1.1 Hardware Specifications

Our experiments were conducted on a dedicated Ubuntu 22.04 server configured with 128 GB of RAM. This environment was essential for efficiently running large language models (LLMs) and performing the computationally intensive tasks involved in generating UML class diagrams.

Interfacing with Local Language Models

To manage and run LLMs locally, we employed the Ollama platform. Ollama is a versatile framework that simplifies the deployment and interaction with models such as llama3.3 and deepseek-r1. It supports cross-platform compatibility working seamlessly on Linux, Windows and macOS and offers several interfaces:

- Terminal-based commands (ollama run <modelName>)
- HTTP API requests via http://localhost:11434/api/generate
- Integration into custom applications through common programming libraries

For this project, we chose to communicate with Ollama using Python, pairing it with the LangChain library to construct modular, logic-driven workflows. LangChain provides a flexible infrastructure for managing interactions with LLMs, supporting features like prompt engineering, chaining and Retrieval-Augmented Generation (RAG) with ease.

Vector Management for RAG

The RAG process required a solution for storing and accessing vector embeddings generated from segmented text inputs. For this, we used Chroma, an open-source vector database purpose-built for machine learning and AI workflows. Its native compatibility with LangChain

allowed for seamless integration and efficient management of embedding data throughout the retrieval and generation process.

Experiment Configuration and Workflow Automation

To organize our test scenarios and automate the UML class diagram generation process, we adopted a structured approach similar to that used in the work of Tsilimigkounakis [28]. This involved designing a table where each row represents a distinct test scenario, complete with detailed metadata to guide the generation workflow. By standardizing scenario definitions in this way, we enabled automated processing and ensured consistency, reproducibility and clarity across all experiments.

The table includes the following key fields:

- **ID**: A unique identifier assigned to each scenario, directly linked to the corresponding generated class diagram.
- **familyId**: Represents the broader group or "family" to which the scenario belongs. It also reflects the directory structure. The format follows:
  {architecture}{fr_set_number}{nfr_set_number}{prompt_file_number}.
  For instance, cs111 indicates a Client-Server architecture with functional requirements set 1, non-functional requirements set 1 and prompt file 1.
- **architecture**: Defines the architectural design pattern requested for that scenario (e.g., Client-Server, Three-Tier, Micrservices).
- **frPathfile**: File path to the functional requirements (FR) used in the scenario.
- **nfrPathfile**: File path to the non-functional requirements (NFR).
- **srsPathfile**: File path to the SRS (Software Requirements Specification) file used in the scenario.
- **promptPathfile**: File path to the textual prompt that guides the model.
- **modelMetadata**: Specifies which language model was used, along with its version (e.g., llama3.3:latest, deepseek-r1).
- **source**: Describes how the diagram was generated, either programmatically through Python scripts or manually via online tools like ChatGPT or Gemini.
- **ragDecision**: Indicates whether the scenario utilizes a Retrieval-Augmented Generation (RAG) approach.
- **ragPathfile**: Path to the external RAG data file, used when RAG is enabled.

- **embeddingsMetadata**: A JSON object detailing the configuration and behavior of the RAG process.

- **resultPath**: Automatically filled by the script post-execution, pointing to the generated response file.

By using this structured table, our Python scripts can loop through all defined scenarios, automating the generation of UML class diagrams based on their respective requirements and parameters. This setup supports controlled experimentation across various architectural styles, LLM configurations and RAG applications and ensures all results are reproducible, traceable and organized for further analysis.

3.1.2 Selecting the UML Output Format from LLMs

The effectiveness of large language models (LLMs) in generating UML diagrams largely depends on the output format they produce. When prompting a large language model (LLM) to generate software architecture diagrams in UML, the output can take various forms, including XMI, PlantUML, Mermaid, or image files. We observed that the quality of the generated diagrams varies depending on the output format. This variation arises because LLMs are trained on data that unevenly represent these formats, leading to differing levels of proficiency. Our goal is to continue this research using the output format where LLMs demonstrate the highest quality responses.

For that reason, in the initial phase of our research, we experimented with various output formats for UML class diagram generation using large language models (LLMs). These formats included image-based outputs, XMI (XML Metadata Interchange) and Diagram-as-Code (DaC) approaches such as Mermaid and PlantUML. Our assessment revealed that LLMs consistently delivered superior accuracy and structural integrity when generating diagrams in PlantUML format, which subsequently became our standard output format for all class diagram generation experiments in this study.

It is important to note that the utilization of model-oriented tools like Visual Paradigm presented significant technical challenges for both generation and evaluation processes. Requesting class diagrams in XMI format for Visual Paradigm import posed different obstacles, including complex syntax requirements, platform-dependent variations and potential compatibility issues that often result in errors or incomplete diagrams.

One alternative approach involved working with class diagrams as images, which would have limited us to LLMs capable of image generation and recognition which is a substantial deviation from our core research objectives. This method would have introduced additional complications through reduced accuracy and increased complexity in processing diagram components.

The following table presents a detailed comparison of the four UML output formats examined (PlantUML, Mermaid, Image Output, and XMI) based on four key criteria relevant to their suitability for use with large language models: ease of generation, syntax robustness, post-generation usability, and accuracy in representing class attributes and relationships.

Output Format	Ease of Generation	Syntax Robustness	Post-Generation Usability	Accuracy in Class Attributes and Relationships
PlantUML	Well-structured and intuitive for LLMs to produce.	Moderately robust. Most syntax issues are easy to fix.	Requires code-based editing, lacks visual editing support.	Captures most standard class features but lacks full UML semantic support.
Mermaid	Simpler syntax, but LLMs sometimes confuse Mermaid-specific conventions.	Prone to syntax errors. Small mistakes often break rendering.	Requires code-based editing, lacks visual editing support.	Supports only basic class declarations and relations.
Image Output	Not directly generatable by all LLMs.	Immune to syntax errors as it's a static visual.	Useful for viewing only, not editable or machine-readable.	Accuracy in class attributes and relationships depends entirely on the model's prior exposure to UML visuals, with no underlying semantic or verifiable representation.
XMI	Difficult for LLMs to generate due to its verbosity and strict structure.	Very fragile. Small formatting errors break parsing and are often very difficult to fix.	Easily imported into modeling tools for graphical editing.	Fully supports UML semantics, including advanced constructs and constraints.

3.1.2.1 PlantUML

Following our comparative evaluation, we adopted PlantUML as our preferred solution. This approach combines the capability to create detailed, well-structured UML diagrams with a straightforward, code-like, text-based language that LLMs can efficiently generate and understand. PlantUML's syntax accessibility ensures that any LLM with basic text generation capabilities can effectively produce and recognize diagrams, avoiding the compatibility challenges associated with XML while avoiding the complexities of image recognition processing.

Furthermore, PlantUML offers visualization capabilities across multiple platforms and tools, including an official web server functioning as an editor and dedicated libraries for various programming environments such as Java, Python and React. For our specific implementation, we used PlantUML's Java .jar file, supporting direct diagram rendering and visualization as images.

This functionality significantly enhanced our workflow efficiency by providing a streamlined process for converting PlantUML code to image format, analysis and evaluation of the generated class diagrams, as demonstrated in Figure 3.1, which illustrates the relationship between PlantUML code and its corresponding UML diagram representation.

```
@startuml

class Company {
  - name: String
  - location: String
  + getDetails(): String
}

class Employee {
  - name: String
  - employeeId: int
  - role: String
  + getProfile(): String
}

class Project {
  - projectName: String
  - deadline: Date
  + getStatus(): String
}

' Associations
Company "1" -- "many" Employee : employs >
Employee "many" -- "many" Project : works on >
Company "1" -- "many" Project : owns >

@enduml
```

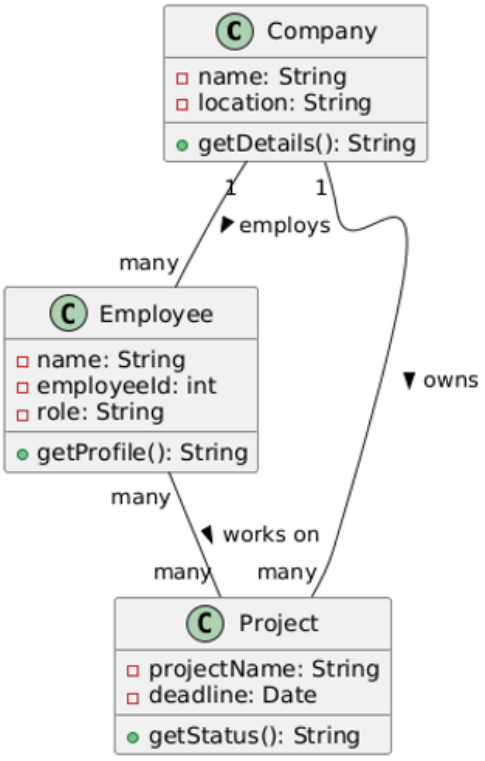


Figure 3.1: Example of PlantUML Code

3.1.3 Retrieval Augmented Generation (RAG)

3.1.3.1 RAG Overview

Retrieval-Augmented Generation (RAG) represents a significant advancement in large language model (LLM) technology, designed to address fundamental limitations in these systems. While LLMs demonstrate remarkable language capabilities, they operate by identifying statistical patterns in language rather than achieving true semantic comprehension. This limitation often results in information that appears convincing but may be factually incorrect or outdated, a phenomenon commonly referred to as "hallucination". RAG frameworks mitigate this issue by integrating external knowledge retrieval into the generation process.

At its core, RAG operates through a two-phase process: first retrieving relevant information from external, verified knowledge sources, then incorporating this information into the language generation process. This approach effectively grounds the model's responses in documented facts rather than relying solely on learned parameters. When a query is submitted, the RAG system identifies and retrieves relevant information from its knowledge base, which then serves as supplementary context for the LLM as it formulates a response. This process creates a bridge between the statistical reasoning capabilities of LLMs and verified external knowledge.

The implementation of RAG offers several significant advantages in production environments. Most notably, it substantially improves response accuracy and reliability by relating outputs to factual information. Additionally, RAG systems provide traceability through citation capabilities, allowing users to verify the sources underpinning generated content, a critical feature for building trust in applications requiring high information precision and reliability. From a practical perspective, RAG reduces the frequency of model retraining required to maintain relevance, as new information can be incorporated by simply updating the external knowledge base rather than retraining the entire model.

From a technical standpoint, RAG frameworks typically use complex information retrieval techniques such as vector embeddings to identify semantically relevant content, which is then injected as contextual information alongside user queries. This contextual enrichment transforms the LLM's operational environment, providing it with precise, relevant information directly applicable to the current query. These systems carefully balance the computational cost of retrieving context with improvements in output quality, allowing them to achieve notable increases in factual accuracy without sacrificing response times. As LLM applications expand into domains requiring precise factual information, RAG has emerged as an essential process for responsible AI system development that uses external context to enhance model capabilities.

3.1.3.2 Embeddings in RAG Systems

Embeddings form the mathematical backbone of modern Retrieval-Augmented Generation (RAG) systems by representing text as dense vectors that capture its underlying meaning. Unlike traditional keyword matching, embeddings focus on the concepts behind the words, placing similar ideas close together in vector space, even if the exact wording differs. This ability to retrieve information based on meaning rather than exact words makes embeddings especially powerful for RAG, greatly enhancing the system’s capacity to find contextually relevant information.

In RAG architectures, the embedding process occurs in two critical phases. Initially, during knowledge base preparation, each document or text segment is transformed into a vector through specialized neural network models trained specifically for semantic representation. These document embeddings are indexed in vector databases optimized for similarity search operations. Subsequently, during the retrieval phase, user queries undergo the same embedding

transformation, creating query vectors that occupy the same semantic space as the document embeddings.

The retrieval mechanism operates on vector similarity principles, typically using distance metrics such as cosine similarity or Euclidean distance to identify the closest conceptual matches between query and document embeddings. Modern implementations often utilize approximate nearest neighbour algorithms to efficiently search vast vector spaces containing millions of document embeddings while maintaining low latency characteristics. This approach allows RAG systems to rapidly identify and retrieve the most semantically relevant contextual information from extensive knowledge bases.

The effectiveness of RAG systems is linked to embedding quality, with more advanced embedding models capturing increasingly complicated semantic relationships. Recent advancements have produced embedding models capable of representing complex relationships between entities, understanding domain-specific terminology and preserving hierarchical concept structures. When paired with efficient vector indexing technologies, these advanced embeddings allow RAG systems to provide highly relevant contextual information to language models, improving response accuracy while maintaining the natural fluency characteristic of modern LLMs.

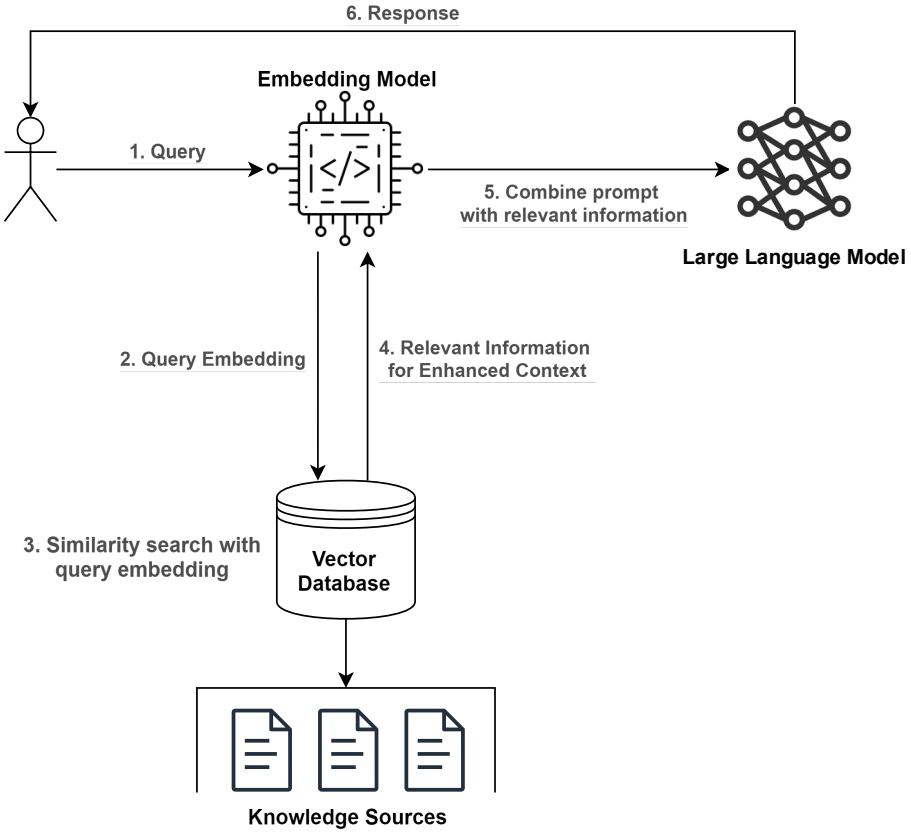


Figure 3.2: RAG Method Illustrated

3.1.3.3 RAG Techniques

In the following section, we explore various RAG techniques, outline the available options and explain the specific RAG pipeline chosen for our experiments, along with the rationale behind that choice.

Embedding Models

The quality of embeddings directly determines retrieval accuracy in RAG implementations, making model selection a critical decision. One can choose between four primary deployment options: self-hosting open-source models, utilizing cloud providers offering open-source implementations, employing proprietary embedding services (e.g., OpenAI), or implementing integrated end-to-end solutions with built-in embedding functionality.

When choosing embedding models for Retrieval-Augmented Generation (RAG) applications, four key factors should guide the decision. First, *Retrieval Average Performance* measures how well a model performs on retrieval tasks using standard benchmarks. Second, *Model Size* and *Memory Usage* impact computational demands, while larger models often provide better accuracy they also require more resources. Third, *Embedding Dimensions* that affect the model's ability to capture subtle semantic relationships and the storage needed, which means that higher dimensions offer better accuracy but increase computational load. Finally, *Maximum Token Length* defines the amount of text that can be processed in a single embedding, influencing how documents need to be divided and handled.

The optimal embedding model varies by use case, with domain-specific applications often benefiting from specialized models while general knowledge implementations may require broader models with higher parameter counts. Organizations should evaluate models using representative data from their knowledge domain rather than relying solely on published benchmarks to ensure selection of embeddings that balance performance with implementation constraints.

Chunking Methods

Chunking is the strategic decomposition of documents into smaller units that enhance retrieval efficiency and precision in RAG systems. This process determines how information is indexed and retrieved, directly impacting the contextual relevance of generated responses. Effective chunking balances context preservation with computational efficiency, creating segments that contain sufficient information while maintaining reasonable storage requirements. RAG implementations typically employ four primary chunking methodologies.

- **Sliding Window Chunking**: Creates overlapping segments of predetermined size, preserving context across boundaries at the cost of storage redundancy

- **Document-Based Chunking**: Respects natural document structures like paragraphs or sections, maintaining inherent information organization but potentially creating inconsistent chunk sizes

- **Semantic Chunking**: Identifies conceptual boundaries within text, grouping related information regardless of structural indicators

- **Agent-Based Chunking**: Optimizes segments for distributed processing across specialized retrieval agents in multi-agent systems

The optimal chunking approach varies based on content characteristics and application requirements. Technical documentation often benefits from document-based strategies, while conversational content typically performs better with sliding window techniques. Implementation parameters such as chunk size and overlap percentage require manual optimization through performance testing with representative queries. By evaluating different chunking techniques, RAG systems can achieve an optimal balance between retrieval accuracy, contextual coherence and computational efficiency.

Although there are additional techniques that could enhance the RAG process such as metadata filtering, query transformation and reranking, our experimentation has determined that these approaches fall outside the current scope of our work.

3.1.3.4 Our RAG Pipeline

Building on the work of Tsilimigkounakis [28], we excluded underperforming RAG methods such as the recursive chunking technique from our scope. Furthermore, we chose to proceed exclusively with the *nomic-embed-text embedding* model from Nomic AI, as it demonstrated comparable performance to OpenAI's text-embedding-3-large. Given that our focus is not on exhaustively experimenting with various RAG methods, we narrowed our approach to using nomic-embed-text in combination with semantic chunking.

The documents inserted into the vector database for each experiment will be referenced later in this work within the corresponding experiment descriptions.

3.2 The DCC Experiment, revisited

In this experiment, we explore whether Large Language Models (LLMs) benefit more from Software Requirements Specification (SRS) documents compared to basic lists of Functional and Non-Functional Requirements. To do so, we replicated the original experiment, from Tsilimigkounakis [28] work, using an SRS document as the input format for software requirements, replacing the simpler requirements lists. Additionally, we excluded RAG methods or materials that previously demonstrated poor performance and further investigation had limited research value. This study seeks to shed light on whether LLMs can better comprehend and utilize richer, more detailed software descriptions, as opposed to minimal input formats.

3.2.1 Architectures Considered

The architectural patterns considered such as Client-Server, Three-Tier and Model-View-Controller (MVC) remain unchanged, as the primary focus of this experiment is to compare SRS documents with FR-NFR lists. Introducing additional architectural patterns would dilute the objective and compromise the clarity of the comparison.

3.2.2 Case Study: DCC (Dummy Coordinate Converter) Application

The Dummy Coordination Conversion (DCC) Application is a software system designed to manage coordinate groups in both Cartesian and polar formats. It allows users to convert, store, retrieve, modify and delete coordinate groups.

Originally introduced in previous work [28], the application remains unchanged in this study. Its simplicity and straightforward functionality make it an ideal candidate for evaluating the AI's ability to adhere to a specified architectural pattern based on clearly defined requirements. By reducing the complexity of the use case, we can more effectively assess the AI's architectural and diagrammatic accuracy.

Software Requirements Specification Input

Given the uncertainty around how Large Language Models (LLMs) would respond to software descriptions provided through a Software Requirements Specification (SRS) document, we designed an experiment using two variations: a concise version and a more extensive one.

The concise SRS (Appendix A (SRS_v1 for DCC Application)) focuses on the essential elements of the application, including use case diagrams and lists, activity diagrams for each use case and data requirements. In contrast, the extensive SRS (Appendix B (SRS_v2 for DCC Application)) converges to what would typically be found in a real-world specification. It includes a detailed system overview, business context, the intended user base, system scope, suggested technologies and constraints and any assumptions or dependencies.

This approach allows us to assess whether LLMs perform better when given richer contextual and business-level information, as opposed to a minimal functional specification.

3.2.3 The Prompt

We maintained the original prompt structure used in previous experiments, modifying only the section where the FR-NFR lists were replaced with the SRS document. This decision was made to preserve the integrity of the comparison. Introducing a new prompt format could have introduced confounding variables, compromising the clarity of the results.

> Task Description: You are tasked with processing a software description and its requirements to generate a class diagram using PlantUML. The class diagram should respect the requested software architecture, define all necessary classes, and accurately represent associations between them.
>
> The requirements for the software we want you to design are in the SRS (Software Requirements Specification) here: {**SRS**}
>
> The demanded Architecture is {Architecture}
>
> Generate a class diagram using PlantUML that defines all necessary classes and associations based on the described architecture, requirements, and functionality. Ensure that:
>
> The diagram reflects the separation of concerns based on the requested architecture.
> Each class is properly defined with attributes and methods.
> All associations (e.g., composition, aggregation, or inheritance) between the classes are included in PlantUML and are clearly represented.
> Create the appropriate packages to include the classes.

3.2.4 LLM Selection

When selecting LLMs for this experiment, we made a conscious decision to retain the models used in previous work. This allowed for a controlled comparison, isolating the effect of replacing the FR-NFR lists with an SRS-based application description while keeping all other variables constant. Additionally, we included several newer models available at the time of the study such as o1, o1-mini, o3-mini-high and deepseek-r1, to evaluate their performance within the same experimental framework.

Model	**Source**	**Parameter Size**	**Quantization**
llama3.1	local	8 B	Q4_0
deepseek-r1:70b	local	70.6 B	Q4_0

Model	Source	Parameter Size	Quantization
command-r	local	32.3 B	Q4_0
gemma2:27b	local	27.2 B	Q4_0
mixtral:8x7b	local	47.6 B	Q4_0
phi3:medium-128k	local	14 B	Q4_0
claudeSonnet3.5	online	N/A	N/A
deepseek-r1	online	671 B	N/A
gemini-1.5	online	N/A	N/A
gpt4o	online	N/A	N/A
gpt4oSAV	online	N/A	N/A
o1	online	N/A	N/A
o1-mini	online	N/A	N/A
o3-mini-high	online	N/A	N/A

3.2.5 RAG Material

The RAG file used in this experiment comes from a well-established source: *Software Engineering* (10th Edition) by Ian Sommerville. This textbook is widely recognized in the field and offers in-depth coverage of core software engineering concepts. By using material from such a trusted source, we aim to give the model reliable information that can help it generate more accurate and structured class diagrams. Including content from Sommerville also lets us explore how academically solid material influences the model's understanding of software architecture.

To make the RAG even more effective, we organized the documentation by architectural style, creating a separate file for each one. This way, we can match the RAG file directly to the

architecture type requested in the prompt. The goal is to see if giving the model focused, architecture-specific guidance improves its ability to follow the desired pattern. By isolating the content this way, we can better assess whether targeted documentation works better than a single, combined resource.

This RAG material approach, splitting the RAG file into three distinct documents, one for each architecture type, follows the second method from Tsilimigkounakis' work [28], which demonstrated the highest performance. By adopting the same strategy and reusing the corresponding RAG documents, we ensure a valid and consistent comparison framework for our experiment.

3.2.6 Evaluation Process

The final phase of our experiment focuses on a evaluation of the generated class diagrams by human experts. The evaluation is based on a set of predefined criteria, which remain unchanged from the previous experiment to ensure consistency and enable a valid comparison of results.

Evaluation Criteria

- **Adherence to Architecture:** Human experts evaluate the extent to which each class diagram conformed to the requested architectural pattern. This involves examining whether core architectural principles are followed, including the proper distribution of responsibilities across classes and structural alignment with the designated architecture (e.g., MVC, Three-Tier, Client-Server).
- **Correctness of Class Relationships**: This criterion focuses on the accuracy of inter-class relationships in the context of the given architecture. Experts assess whether associations, dependencies and communication flows are correctly modeled and whether they comply with architectural best practices.
- **Cohesion and Coupling:** Diagrams are evaluated on their ability to achieve high cohesion and low coupling, two foundational principles of sound software architecture. High cohesion is judged by how focused and purpose-driven each class is, while low coupling is assessed based on the degree of independence between classes.
- **Consistency with Software Requirements:** Ensure that the diagrams accurately reflect both the functional and non-functional requirements defined in the input specifications. This ensures that beyond structural correctness, the diagrams capture the full scope of expected system behavior.

3.2.7 Scenarios Performed

To understand how different factors like model choice and the use of RAG affect the quality of AI-generated class diagrams, we ran a series of experiments. We designed a range of scenarios to help isolate and analyze the impact of each variable. While staying at the scope of

architectures such as Client-Server, Three-Tier and MVC we varied a combination of parameters resulting in 120 scenarios.

- **Large Language Models (LLMs):** We used different models to evaluate how model selection impacts the quality of generated architecture.

- **RAG vs No-RAG:** We tested scenarios with and without Retrieval-Augmented Generation (RAG) to examine how access to supplementary information affects the accuracy and structure of the generated diagrams.

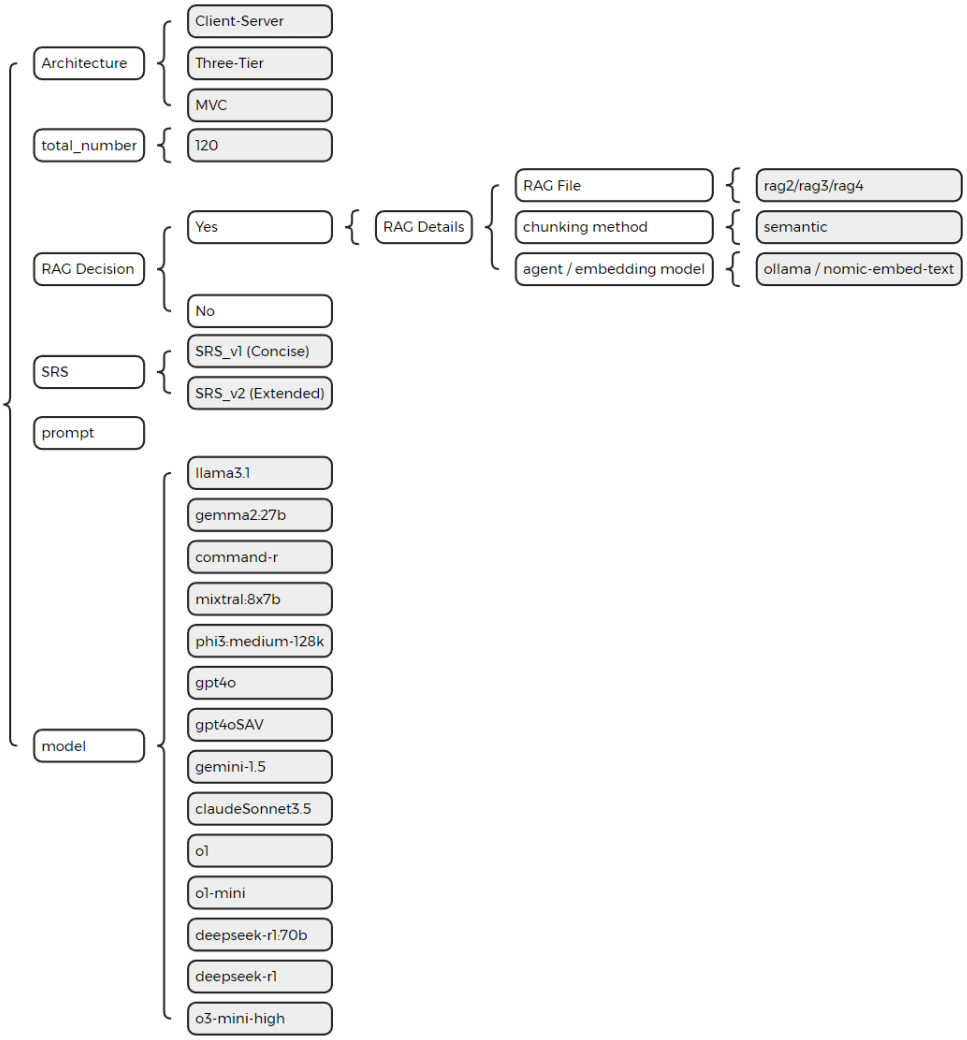


Figure 3.3: Scenarios Performed - DCC

3.2.8 Reference Architectures

In this section, we present reference class diagrams for the proposed architectures for DCC application. These reference architectures we inherited from Tsilimigkounakis work [28]. It is important to state that these reference architectures were not used as a benchmark to evaluate

the generated diagrams, based on similarity. Instead, it is included to illustrate the type of architectural outputs we considered a strong response from the language models in this experiment.

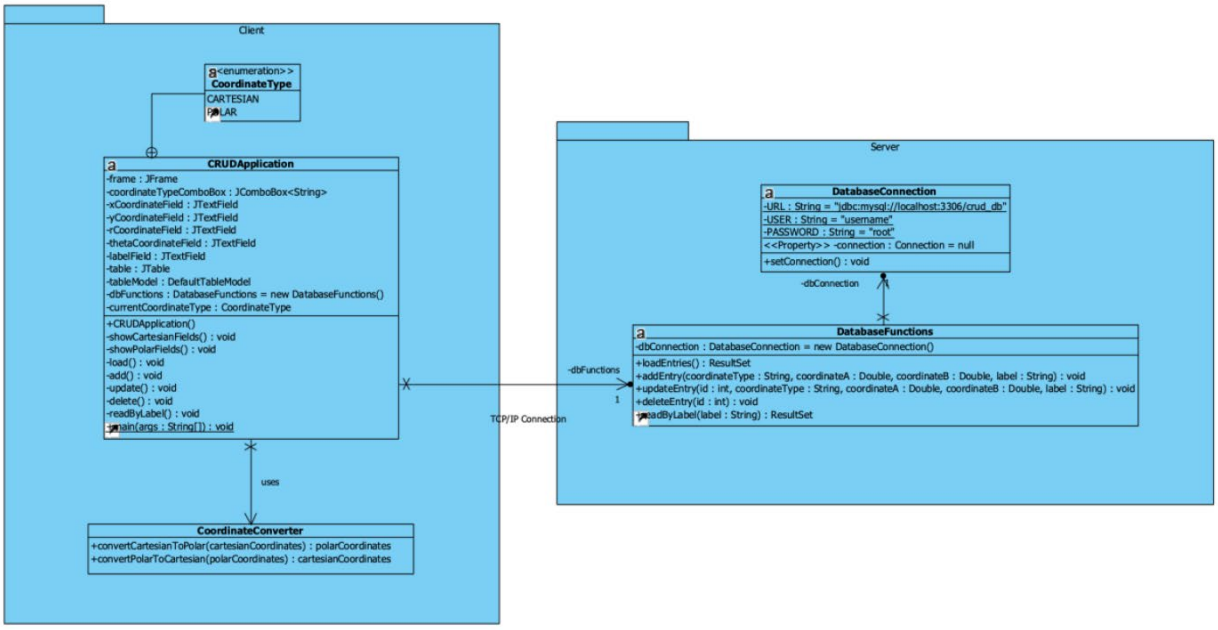


Figure 3.4: Client Server Architecture (DCC App)

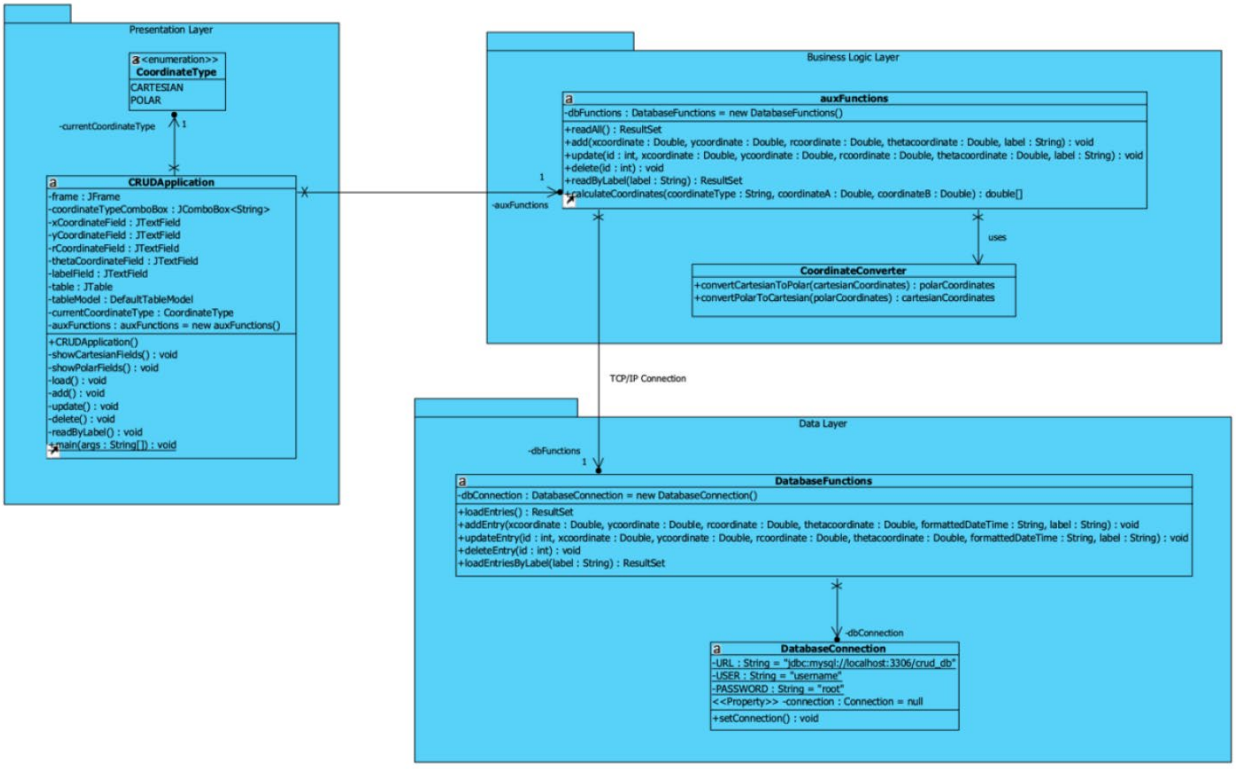


Figure 3.5: Three-Tier Architecture (DCC App)

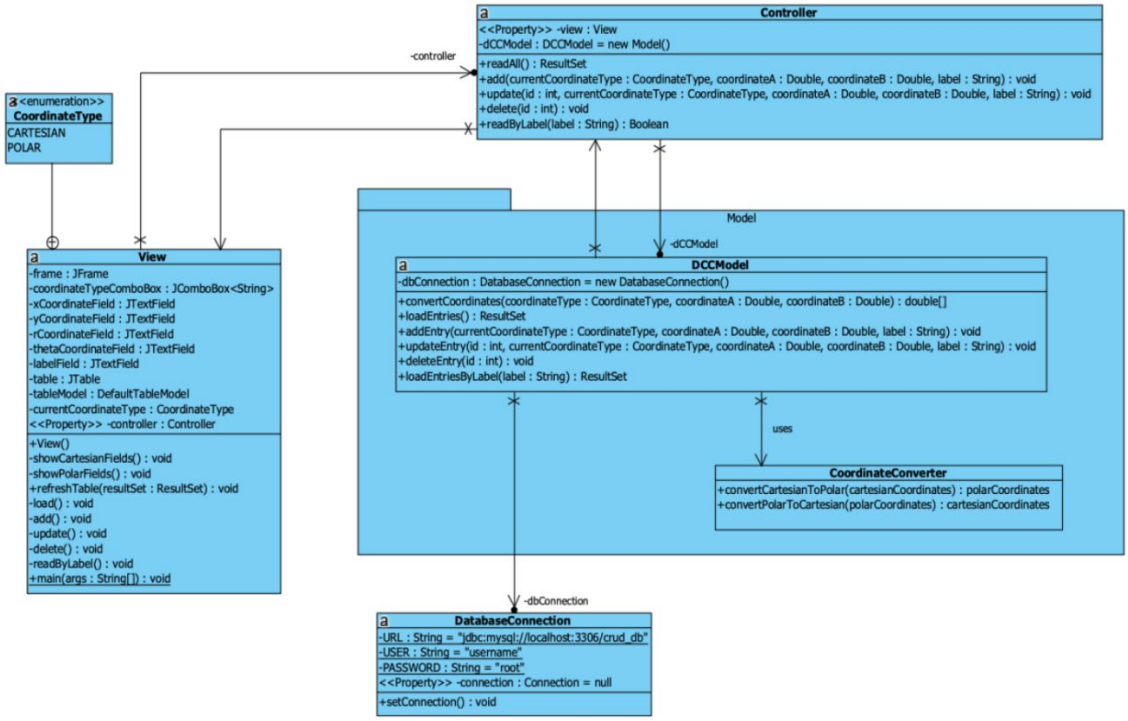


Figure 3.6: MVC Architecture (DCC App)

3.3 The MyCharts Experiment

In this section, we present the experiment conducted to evaluate the generation of class diagrams by AI for a more complex application with requirements of moderate complexity, an area that represents the core focus and primary research interest of this study. We describe the application's functional and non-functional requirements, outline the Software Requirements Specification (SRS) document and detail all relevant experimental parameters, including the RAG materials, the models selected, the prompt, evaluation criteria for the human evaluation, the metrics for objective evaluation and a reference architecture.

3.3.1 Architectures Considered

For this experiment, we chose to focus exclusively on the Microservices Architecture, applying a new set of evaluation criteria specifically to this architectural paradigm. Since the primary objective was to assess the capabilities of AI in generating a more complex system architecture, revisiting the previously explored architectures was deemed unnecessary.

3.3.2 Case Study: MyCharts Application

MyCharts Application is a web-based service designed to allow users with minimal technical expertise to generate, manage and download charts in various formats. It simplifies chart creation by providing templates for source data, supporting data uploads. It uses the Highcharts library for chart generation.

The application allows users to:
- Download CSV templates for supported chart types.
- Upload CSV files to generate charts.
- Save and download charts in PDF, PNG, SVG and HTML formats.
- Purchase quotas for chart creation.
- View and download generated charts.

The selected software application presents a higher level of complexity compared to the DCC Application examined in the previous experiment. It was inherited from the “Software-as-a-Service Technologies” course at NTUA, where it served as a term project. Its design and requirements are sufficiently advanced to justify the application of the Microservices Architectural paradigm, providing a more realistic and demanding scenario for evaluation. At the same time, the complexity remains manageable, ensuring that it does not exceed the capabilities of the AI system to produce meaningful and high-quality software diagrams.

In this experiment, we tested two different types of input to evaluate which would enable the AI to produce higher-quality software architecture class diagrams. The first input type consisted of a list of functional and non-functional requirements written in natural language. This input provided only the essential information necessary for the application's design, without

offering deeper insights into its business logic or operational details. The second input type was a complete Software Requirements Specification (SRS) document, which included more detailed information about the application, such as business logic descriptions, use case diagrams, activity diagrams and additional contextual insights.

Functional and Non-Functional Requirements Input

Below are the functional and non-functional requirements used to describe the MyCharts Application. This information was included as part of the prompt in the AI experiment and served to define the application for which the software architecture was to be generated.

Functional Requirements

1. Authenticate users via Google accounts.
2. Allow download of CSV templates for 3 supported chart types (Basic line, Line with annotations, Basic column).
3. Upload CSV file with the user data for chart generation.
4. Validate uploaded CSV files against the template structure, data types, and mandatory data existence.
5. Generate charts with data from the CSV file uploaded by the user, using Highcharts.
6. Save charts to the server in PDF, PNG, SVG, and HTML formats.
7. Display a dashboard with the history of user-generated charts, including chart previews.
8. Download selected chart type.
9. Charge quotas for chart creation.
10. Allow users to delete or download charts.
11. Sell quotas and receive payment by a payment gateway
12. Maintain user profiles containing the following data: Name (from Google account), profile picture (from Google account), email (from Google account), remaining quota, account creation timestamp, last login timestamp.
13. Display user's profile info, including remaining quotas.

Non-Functional Requirements

Performance
- Chart generation completes within 3 seconds.

Usability
- Intuitive UI with guided workflows (e.g., tooltips, validation hints) to assist non-technical users.

Security
- SSL encryption: Data transmitted over the internet is encrypted using SSL.

Availability
- The service should be available 90% of the time daily.
- The system should prioritize Availability over Consistency during network partitioning.
- While in network partitioning, although temporary data inconsistencies may occur, as many as possible services should remain fully operational.

Scalability
- Support 1,000 concurrent users.

It is important to note that many of the non-functional requirements cannot be directly evaluated through the PlantUML-generated design, as their verification would require full application development and testing. Nevertheless, we included them to create a more realistic use case scenario, simulating the workflow a software engineer might follow. Where, after obtaining the initial software architecture in UML form, they would continue prompting the AI to generate code and progressively build and test the complete application. While the scope of this study, is restricted to the generation of software architecture, we believe including this aspect provides more realistic insights.

Software Requirements Specification Document

The Software Requirements Specification (SRS) document used to describe *MyCharts* Application (Appendix C (SRS for MyCharts Application)) was included as part of the prompt in the AI experiment and served as the basis for defining the application for which the software architecture was to be generated.

The Software Requirements Specification (SRS) contains all the essential information needed to design the architecture, while also including additional details that go beyond the immediate scope of the task. However, these supplementary elements were retained to simulate a more realistic use case, one in which a software engineer uses an AI model to derive the initial architecture directly from a SRS document.

3.3.3 Evaluation Process

This experiment includes an evaluation of the generated class diagrams, against carefully selected criteria specific to the unique characteristics and principles of the Microservices Architectural Paradigm. This ensures that the assessment reflects not just general diagram quality, but also alignment with principles of Microservices design.

These evaluation criteria differ from those used in previous experiments, as earlier benchmarks lacked the specificity needed to properly reflect the principles and demands of the Microservices architecture. Unlike more traditional architectures such as Client-Server, Three-Tier, or Model-View-Controller, Microservices present a significantly higher level of complexity and require a more detailed set of evaluation criteria to validate proper design and alignment with best practices.

Evaluation Criteria

1. **Functional Alignment & Responsibility Distribution:** Ensure that each microservice maps to a bounded context and implements a focused set of functionalities. The union of the functionalities of each microservice should be the entire set of functional requirements.

2. **Coupling & Deployment Independence:** Ensure that microservices are loosely coupled and independently deployable

3. **Cohesion:** Ensure high cohesion meaning that more than one microservice needs to be involved in the completion of each use case.

4. **Data Management:** Ensure that each service is responsible for managing specific data elements without using shared databases.

5. **Data Consistency:** Ensure that operations to achieve eventual consistency of data states among different microservices are included.

6. **Communication & Flow Control:** Ensure that service coordination is done using choreography or orchestration mechanisms. API gateways and/or messaging (pub-sub) services should be used to implement flow control.

7. **Non-Functional Requirements:** Ensure that the design satisfies all problem-specific non-functional requirements.

Human experts with varying software architecture expertise evaluated the diagrams. Each reviewer independently rated the diagrams from 0-5 across the 7 criteria. This approach provided qualitative feedback for quality assessment.

3.3.4 Metrics Considered for Objective Evaluation

A focus of this research was finding an objective way to evaluate AI-generated architectures using solid metrics rather than just subjective evaluations. With AI technology evolving so quickly, we need a reliable system to measure how these models are improving over time regarding software design capabilities. We realized that the metrics depend heavily on what type of architecture we're evaluating, therefore, different architectural styles need different metrics.

We selected which metrics to use based on previous work regarding evaluating Microservices architectures. We were guided by Engel's work [35] on evaluating already implemented and deployed microservices applications and Bogner's significant contributions [36],[37] that focused on theoretically evaluating a system design, designed using microservices. Their research helped us identify what really matters when evaluating microservice designs. Additionally, we introduced new metrics that we deemed essential to the evaluation process.

Based on this prior work, we carefully selected metrics that capture the most important aspects of good microservices architecture: how well services stick to single responsibilities, how many interfaces each microservice has, how they communicate with each other, how well they align with business domains and whether they can be deployed independently etc. Additionally, we opted to structure a conceptual corelation between the quantitative evaluation

metrics (objective) and some of the qualitive evaluation criteria (subjective) evaluated by human experts.

We selected the following metrics to evaluate the quality of AI-generated microservices architectures:

1. Metrics applicable across the entire architecture

- **SI (Statelessness Index):**

$$SI = (\#\ of\ stateless\ services)/|Number\ of\ Services|$$

The SI metric measures the proportion of services that remain stateless across requests, i.e., they do not retain a persistent state between interactions. Higher values (closer to 1) suggest that the architecture predominantly consists of stateless services, which enhances scalability, supports containerization and allows for independent deployment.

- **DOC (Data Ownership Count):**

$$DOC = (\#\ of\ services\ with\ a\ dedicated\ data\ store)/|Number\ of\ services|$$

The DOC metric captures the percentage of services that manage their own dedicated data stores. Values approaching 1 indicate strong data autonomy, reflecting a key microservices principle where each service owns its data, reducing coupling and enhancing maintainability.

- **SST (Service Support for Transactions):**

$$SST = (\#\ of\ transaction\ aware\ services)/|Number\ of\ services|$$

The SST metric reflects the percentage of services that implement at least one mechanism for ensuring eventual consistency across distributed data states. Higher values imply that a greater number of services are aware of and actively manage cross-service consistency.

2. Service-specific metrics

- **SIC (Service Interface Count):**

$$SIC(S) = \sum operations\ of\ service\ S$$

The SIC(S) metric measures the number of operations exposed by a given service, S. It serves as an indicator of interface complexity. Lower but adequate values suggest that

the service has a well-defined, focused responsibility, avoiding functional overload and promoting clear separation of concerns within the architecture.

- **AIS (Absolute Importance of the Service):**

$$AIS(S) = \sum other\ services\ that\ invoke\ service\ S$$

The AIS(S) metric represents the number of distinct clients that invoke at least one operation of service S. A balanced distribution of AIS values across services indicates that no single service plays a disproportionately central role, which is essential for maintaining low coupling and ensuring deployment independence within a microservices architecture.

- **ADS (Absolute Dependence of the Service):**

$$ADS(S) = \sum other\ services\ that\ are\ invoked\ by\ service\ S$$

The ADS(S) metric captures the number of other services that service S depends on. Specifically, the number of services from which S invokes at least one operation. A balanced distribution of ADS values across the system suggests an even spread of dependencies, minimizing the risk of bottlenecks and promoting loose coupling and independent deployment.

3. Use-case-specific metrics

- **SC (Service Cohesion):**

$$SC(U) = \{Set\ of\ services\ involved\ for\ use\ case\ U\}$$

The SC(U) metric identifies the set of services that participate in fulfilling use case U. A high number of collaborating services for a single use case indicates high cohesion within the system, which is an expected characteristic of a well-structured microservices architecture, where functionality is distributed.

The following outlines the conceptual relationship between the objective metrics and the evaluation criteria used to assess the generated architectures.

Evaluation Principle	**Metric(s)**
Functional Alignment & Responsibility Distribution	SIC
Coupling & Deployment Independence	SI, AIS, ADS
Cohesion	SC
Data Management	DOC
Data Consistency	SST

While a conceptual alignment between the evaluation principles and the selected metrics is expected, the actual correlation must be validated experimentally. For example, architectures generated with a DOC metric value closer to 1 are expected to achieve higher scores in data management.

Lastly, not all evaluation principles have a corresponding metric, as some criteria are not easily quantifiable. Specifically, aspects such as 'Communication & Flow Control' and 'Non-Functional Requirements' are difficult to assess directly from a UML diagram with any accuracy.

3.3.5 The Prompt

In our experiment, we used two types of prompts: one based on a Software Requirements Specification (SRS) and another based on a Functional and Non-Functional Requirements (FR-NFR) description. Each prompt is structured into the following sections:

- **General Task Description**: A high-level overview of the task assigned to the language model, outlining the overall objective.
- **Application Description**: A detailed description of the target software system for which the architecture is to be generated. This can be provided either in the form of a SRS document or as separate lists of functional and non-functional requirements.
- **Design Guidelines for Microservices Architecture**: A set of best practices and architectural principles for Microservices design, intended to guide the language model toward generating sound and maintainable solutions.
- **PlantUML Design Requirements**: Specific instructions regarding the structure of the class diagram, including attributes, methods, associations and package organization. These directives help ensure that the generated diagrams are syntactically correct and semantically meaningful.
- **Operation Classification in PlantUML Class Diagram**: Instructions for categorizing each class operation in the architecture as one of the following: business logic, data management, data consistency, or flow control. This classification supports the evaluation of architectural quality.
- **Description of Metrics to Be Calculated**: An overview of the metrics to be automatically extracted from the generated diagrams, along with a brief explanation of their purpose and calculation method.
- **Expected Metric Output Format (JSON)**: The required JSON format for presenting the calculated metrics, ensuring consistency and enabling automated analysis.

The following are the two precise prompts employed in our experiment:

Task Description: You are tasked with processing a software description and its requirements to generate a class diagram using PlantUML. The class diagram should respect the requested software architecture, define all necessary classes, and accurately represent associations between them.

The requirements for the software we want you to design are in the SRS (Software Requirements Specification) here: {**SRS**}
The demanded Architecture is Microservices

Important guidelines for designing proper microservices architecture:
1. Functional Alignment & Responsibility Distribution
Focus: Ensure that each microservice maps to a bounded context and implements a focused set of functionalities. The union of the functionalities of each microservice should be the entire set of functional requirements.
2. Coupling & Deployment Independence
Focus: Ensure that microservices are loosely coupled and independently deployable
3. Cohesion
Focus: Ensure high cohesion meaning that more than one microservice needs to be involved in the completion of each use case.
4. Data Management
Focus: Ensure that each service is responsible for managing specific data elements without using shared databases.
5. Data Consistency
Focus: Ensure that operations to achieve eventual consistency of data states among different microservices are included.
6. Communication & Flow Control
Focus: Ensure that service coordination is done using choreography or orchestration mechanisms. API gateways and/or messaging (pub-sub) services should be used to implement flow control.
7. Non-Functional Requirements
Focus: Ensure that the design satisfies all problem-specific non-functional requirements.

Generate a class diagram using PlantUML that defines all necessary classes and associations based on the described architecture, requirements, and functionality. Ensure that:
The diagram reflects the separation of concerns based on the requested architecture.
Each class is properly defined with attributes and methods.
All associations (especially operation calls for service offerings, service dependencies) between the classes are included in PlantUML and are clearly represented.
Create the appropriate packages to include the classes.

Classify each operation into one of the following groups (by commenting next to the operation in the PlantUML):
Functional requirement-business logic operation
Data management operation
Data consistency operation
Flow control operation

Additionally, calculate the following characteristics of the architecture:
General characteristics:
- SI: Counts that services do not carry persistent state across requests and divides with the total number of services. SI = (# of stateless services) / |Total number of services|
- DOC (Data Ownership Count): Counts the services that manage a dedicated data store and divides with the total number of services. DOC = (# of services that are linked to a dedicated data store) / |Total Number of services|
- SST (Service Support for Transactions): Counts the services that have at least one operation classified as "Data consistency operation" and divides with the total number of services. SST(S) = (# of transaction-aware services) / |Total Amount of services|.
Characteristics per service:
  - SIC (Service Interface Count): Counts the interfaces of service S.
  - AIS = Number of distinct consumers of service S
  - ADS = Number of distinct services called by S
Characteristics per use case:
  - SC: Set of services involved for the completion of one-use case. SC = {(Set of microservices involved per use case) MS SET}

Display the characteristics described above, in a JSON object with the following format:
{JSON_METRIC_FORMAT}

Task Description: You are tasked with processing a software description and its requirements to generate a class diagram using PlantUML. The class diagram should respect the requested software architecture, define all necessary classes, and accurately represent associations between them.

The requirements for the software we want you to design described below: {**FR**}, {**NFR**}
The demanded Architecture is Microservices

Important guidelines for designing proper microservices architecture:
1. Functional Alignment & Responsibility Distribution
Focus: Ensure that each microservice maps to a bounded context and implements a focused set of functionalities. The union of the functionalities of each microservice should be the entire set of functional requirements.
2. Coupling & Deployment Independence
Focus: Ensure that microservices are loosely coupled and independently deployable
3. Cohesion
Focus: Ensure high cohesion meaning that more than one microservice needs to be involved in the completion of each use case.
4. Data Management
Focus: Ensure that each service is responsible for managing specific data elements without using shared databases.
5. Data Consistency
Focus: Ensure that operations to achieve eventual consistency of data states among different microservices are included.
6. Communication & Flow Control
Focus: Ensure that service coordination is done using choreography or orchestration mechanisms. API gateways and/or messaging (pub-sub) services should be used to implement flow control.
7. Non-Functional Requirements
Focus: Ensure that the design satisfies all problem-specific non-functional requirements.

Generate a class diagram using PlantUML that defines all necessary classes and associations based on the described architecture, requirements, and functionality. Ensure that:
The diagram reflects the separation of concerns based on the requested architecture.
Each class is properly defined with attributes and methods.
All associations (especially operation calls for service offerings, service dependencies) between the classes are included in PlantUML and are clearly represented.
Create the appropriate packages to include the classes.

Classify each operation into one of the following groups (by commenting next to the operation in the PlantUML):
Functional requirement-business logic operation
Data management operation
Data consistency operation
Flow control operation

Additionally, calculate the following characteristics of the architecture:
General characteristics:
- SI: Counts that services do not carry persistent state across requests and divides with the total number of services. SI = (# of stateless services) / |Total number of services|
- DOC (Data Ownership Count): Counts the services that manage a dedicated data store and divides with the total number of services. DOC = (# of services that are linked to a dedicated data store) / |Total Number of services|
- SST (Service Support for Transactions): Counts the services that have at least one operation classified as "Data consistency operation" and divides with the total number of services. SST(S) = (# of transaction-aware services) / |Total Amount of services|.
Characteristics per service:
  - SIC (Service Interface Count): Counts the interfaces of service S.
  - AIS = Number of distinct consumers of service S
  - ADS = Number of distinct services called by S
Characteristics per use case:
  - SC: Set of services involved for the completion of one-use case. SC = {(Set of microservices involved per use case) MS SET}

Display the characteristics described above, in a JSON object with the following format:
{JSON_METRIC_FORMAT}

The following is the JSON format of the metrics:

```
{
  "SI": "",
  "DOC": "",
  "SST": "",
  "services": [
    {
      "service_name": "",
      "SIC": "",
      "AIS(in)": "",
      "ADS(out)": "",
    },
    {
      "service_name": "",
      "SIC": "",
      "AIS(in)": "",
      "ADS(out)": "",
    },
  ],
  "use_cases": [
    {
      "use_case_name": "",
      "SC": "",
    },
    {
      "use_case_name": "",
      "SC": "",
    },
  ],
}
```

3.3.6 LLM Selection

Based on the outcomes of earlier experiments, we decided to leave out models that had already shown weak performance. Since the current task, designing a Microservices Architecture for *MyCharts* Application, is more complex than the previous DCC application, it didn't make sense to include models that couldn't handle a simpler case effectively.

For this more demanding task, we included a selection of commercially available online models that are generally expected to deliver higher performance. It is important to note that, at the time this research was conducted, models such as DeepSeek-R1, OpenAI's models (e.g., o1, o3-min-high) and Claude Sonnet 3.7 had been released and were incorporated into our study. Since then, additional models have become available, however, in the interest of maintaining a clear scope, we chose to set a boundary for model selection. Further evaluation of newer models is considered a valuable direction for future work.

Model	Source	Parameter Size	Quantization
llama3.3:latest	local	70.6 B	Q4_0
deepseek-r1:70b	local	70.6 B	Q4_0
mistral	local	7.25 B	Q4_0
gemma2:27b	local	27.2 B	Q4_0
mixtral:8x22b	local	141 B	Q4_0
claudeSonnet3.7	online	N/A	N/A
deepseek-r1	online	671 B	N/A
gemini-2.0	online	N/A	N/A
gpt4o	online	N/A	N/A
o1	online	N/A	N/A
o3-mini-high	online	N/A	N/A
mistral-online	online	N/A	N/A
grok3	online	N/A	N/A

3.3.7 RAG Material

In this experiment, we explored two different cases using two distinct RAG (Retrieval-Augmented Generation) files. The first RAG file provides a more abstract and formal overview of the Microservices Architectural paradigm, while the second adopts a more concise and practical approach.

- **RAG1**: *Microservices Patterns* by Chris Richardson, Chapter 2 [38]
- **RAG2**: *Microservices Design Patterns* by Nishant Malhotra, Value Labs [39]

3.3.8 Scenarios Performed

To understand how different factors, like FR/NFR versus SRS input, model choice and the use of RAG, affect the quality of AI-generated class diagrams, we ran a series of experiments. We designed a range of scenarios to help isolate and analyze the impact of each variable.

While staying at the scope of Microservices Architecture we varied a combination of parameters resulting in 46 scenarios.

- **Large Language Models (LLMs):** We used different models to evaluate how model selection impacts the quality of generated architecture.

- **RAG vs No-RAG:** We tested scenarios with and without Retrieval-Augmented Generation (RAG) to examine how access to supplementary information affects the accuracy and structure of the generated diagrams.

- **Functional and Non-Functional Requirements vs SRS:** We experimented with both FR-NFR formatted requirements and full Software Requirements Specifications (SRS) to determine which type of input yields better results.

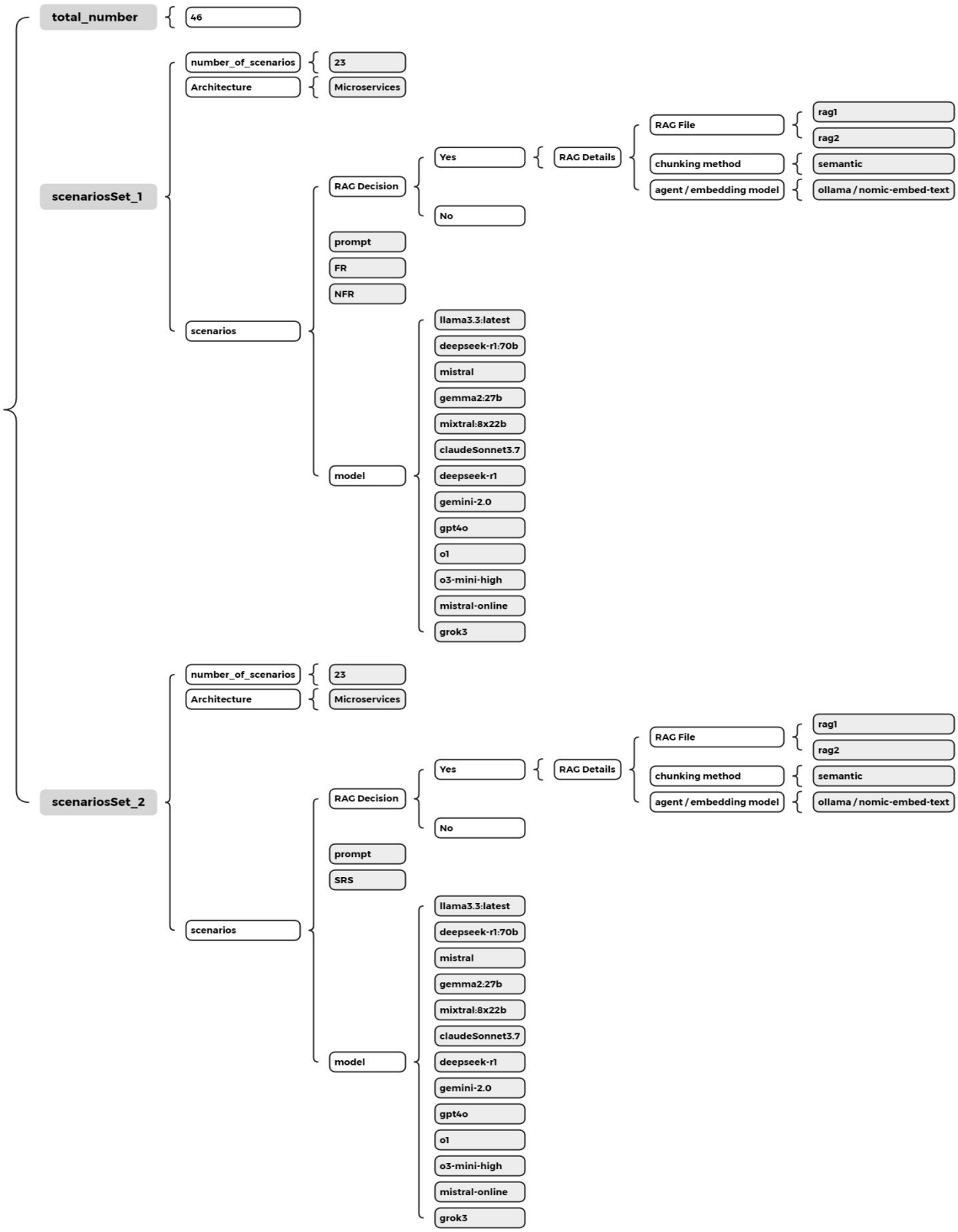


Figure 3.7: Scenarios graph - MyCharts

3.3.9 Experiment Pipeline

In this section, we describe the experiment's pipeline, with a focus on the steps carried out after collecting the generated PlantUML class diagrams that represent the software architecture of the *MyCharts* application, designed following the microservices paradigm. The diagram in Figure 3.8 illustrates the overall workflow of the experiment.

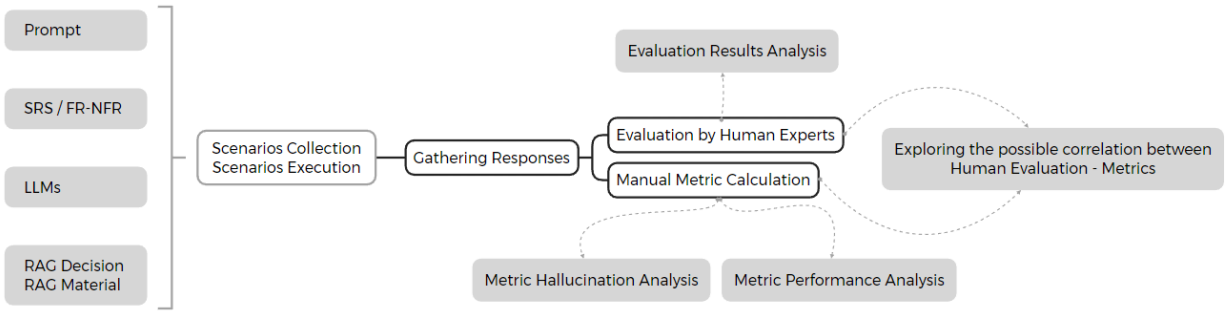


Figure 3.8: MyCharts Experiment Pipeline

Below we describe each step of the experiment pipeline:

- **Scenarios Collection / Scenarios Execution**: In this step, all scenarios outlined in the previous section are compiled into an Excel spreadsheet. Custom scripts are then executed to automate prompt delivery to all LLMs and to organize their responses.

- **Gathering Responses**: All generated outputs are uploaded to a custom Web UI , developed specifically for human evaluation of the class diagrams. This platform provides real-time rendering of PlantUML diagrams and stores evaluator feedback in a dedicated database.

- **Evaluation by Human Experts**: Each team member evaluates the generated class diagrams individually, assigning a score from 0 to 5 across seven predefined evaluation criteria.

- **Evaluation Results Analysis**: Human evaluation data is aggregated and analysed to extract meaningful insights. Graphical visualizations are used to compare performance across different LLMs, RAG configurations and supporting materials.

- **Manual Metric Calculation**: Selected architectural metrics are manually extracted from the generated class diagrams. Both the manually calculated and LLM-generated metrics are stored.

- **Metric Performance Analysis**: Here, we assess the overall quality of each class diagram based on how well its calculated metric performance.

- **Metric Hallucination Analysis**: We compare the manually computed metrics with those automatically provided by the LLMs, assessing the extent to which the models accurately understand and report on architectural metrics.

- **Exploring possible Correlations**: We investigate whether better metric performance corresponds with higher human evaluation scores per principle, offering insight into the potential for objective, metric-driven evaluation of software architectures.

3.3.10 Reference Architecture

In this section, we present a reference class diagram that outlines a proposed architecture for *MyCharts* application. In software engineering, particularly in the design of complex architectures, there is rarely a single "correct" solution. With this understanding, our goal was to provide a generally sound and well-structured example. This reference diagram was not used as a benchmark to evaluate the generated diagrams based on similarity. Instead, it is included to illustrate the type of architectural output we considered a strong response from the language models in this experiment.

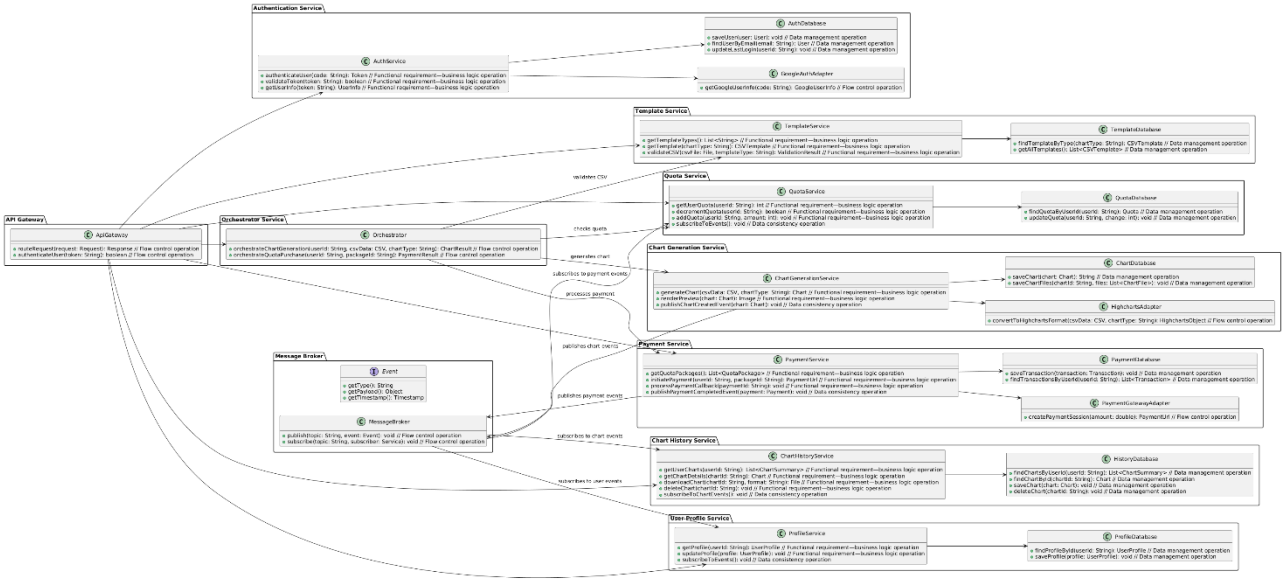


Figure 3.9: MyCharts Reference Architecture

3.4 MyCharts 2-Prompt Experiment

While the primary focus of this research was the *MyCharts* experiment detailed earlier, we subsequently conducted a smaller exploratory test to evaluate whether issuing a second, structured prompt to the LLM could enhance the quality of the generated class diagrams. This conversational approach, which mimics back-and-forth interaction, presents several research limitations, being its reliant on the specific context of the prior response, which makes generalization technically challenging. Despite these constraints, the method holds significant research value, as it mirrors a more realistic scenario in which a software engineer iteratively prompts an LLM to refine architectural outputs. Although limited in scope, this preliminary experiment highlights the potential of this approach and lays the groundwork for future large-scale studies.

3.4.1 Parameters

For this small-scale experiment, we chose to repeat the *MyCharts* experiment using only those LLMs that had previously produced responses of moderate to good quality. The setup remained largely the same: we used the identical prompt from the original *MyCharts* experiment and relied solely on the problem description provided through the Software Requirements Specification (SRS) document. Based on these selection criteria, the following LLMs were included in this follow-up study:

Model	**Source**	**Parameter Size**	**Quantization**
claudeSonnet3.7	online	N/A	N/A
deepseek-r1	online	671 B	N/A
gpt4o	online	N/A	N/A
o1	online	N/A	N/A
mistral-online	online	N/A	N/A

3.4.2 Second Prompt

The strategy behind the second prompt was to introduce a degree of standardization in guiding the LLM toward improving its class diagram design. This guidance was delivered in two main ways. First, we manually computed the metrics for the initial class diagram generated by

the LLM and provided these results as part of the second prompt, explicitly stating that they reflect the first response. Second, we included the human evaluation scores based on the seven criteria outlined in the previous experiment. Finally, we offered standardized guidance encouraging the LLM to regenerate the class diagram with the goal of improving both the evaluation metrics and its performance on the human-assessed criteria. As a result, the follow-up prompt consisted of the following sections:

- Task Description
- Metrics regarding class diagram from first response
- Human evaluation based on first response
- Standardized Guidance

Based on this strategy, the second prompt was formulated as follows:

> Try regenerating the microservices architecture in a PlantUML class diagram with the previous requirements.
> Your current metrics are the following:
>
> {Metrics JSON}
>
> Your current human evaluation for this class diagram is the following:
> Functional Alignment & Responsibility Distribution: X
> Coupling & Deployment Independence: X
> Cohesion: X
> Data Management: X
> Data Consistency: X
> Communication & Flow Control: X
> Non-Functional Requirements: X
>
> Try regenerating a better microservices architecture for the previous requirements achieving a higher human evaluation and better metrics. While re-designing the architecture keep in mind that SI, DOC and SST metrics should be as close to 1 as possible, SC should contain many services per use case and SIC, AIS, ADS metrics should be as low as possible.

3.4.3 Experiment Pipeline

The workflow for this small-scale experiment is illustrated in the diagram in Figure 3.10, providing a visual overview of the sequential steps and components involved in the process.

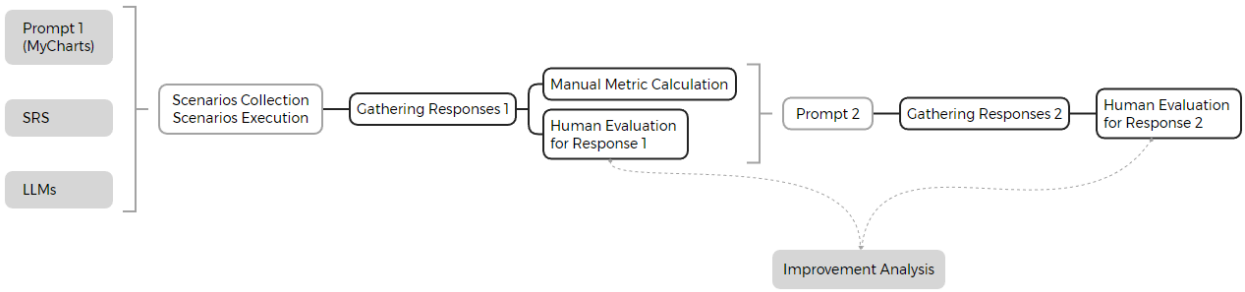


Figure 3.10: 2-Prompt Experiment Pipeline

The pipeline consists of the following procedures:

- **Scenarios Collection:** Gathering all relevant scenarios intended for execution, relying on the selected LLMs used in this experiment.
- **Manual Metric Calculation:** Manually computing the metrics for each generated class diagram.
- **Human Evaluation from Response 1:** Assessing the initial LLM responses using the seven evaluation criteria introduced in the previous experiment.
- **Prompt 2:** Issuing the second, standardized prompt to the LLMs, following the approach described earlier, to guide the regeneration of the class diagram.
- **Human Evaluation for Response 2:** Evaluating the revised class diagrams from the second prompt using the same seven human evaluation criteria.
- **Improvement Analysis:** Analysing whether the second set of responses demonstrated improvement over the first in terms of human-assessed evaluation criteria.

4. Results

4.1 Web Based Evaluation Platform

We enhanced our evaluation methodology for DCC and MyCharts experiments by utilizing the framework initially developed by Tsilimigkounakis [28]. This web application presents generated diagrams through an intuitive, well-structured interface while providing functionality for human evaluation. By expanding this web-based application to accommodate multiple experiments, we established an effective platform for assessing numerous class diagrams and automatically generating visualization graphs across experimental datasets.

In Figure 4.1: Web Based Evaluation Platform - SAAI, Figure 4.2: Web Based Evaluation Platform - SAAI we showcase the Web Based Evaluation Platform that allowed us to simplify the evaluation process store and organize our evaluation results while also generating useful graph visualizations to support analysis and interpretation.

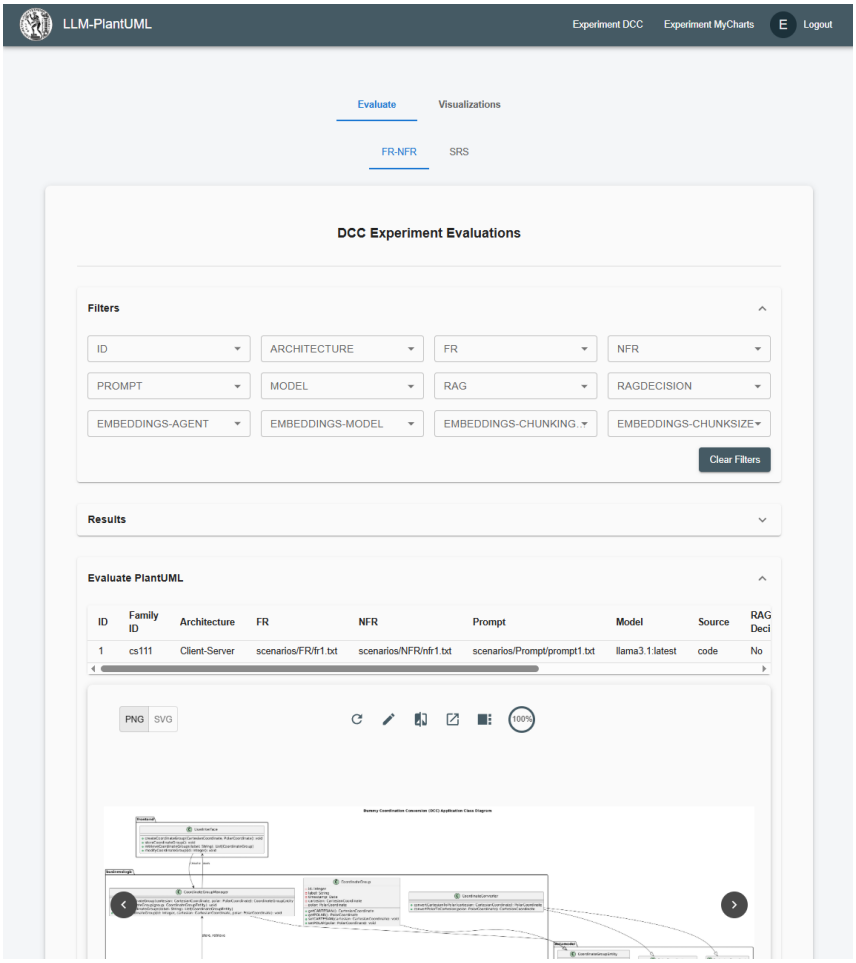


Figure 4.1: Web Based Evaluation Platform - SAAI

Figure 4.2: Web Based Evaluation Platform - SAAI

4.2 DCC Experiment

4.2.1 Typical Cases

In this section, we present a selection of UML class diagrams generated by the LLMs for the DCC experiment. As anticipated, the quality of the diagrams varied significantly, from incomplete and unstructured representations with little relevance to the DCC application (weak diagrams), to well-structured and highly accurate class diagrams that performed exceptionally across all four evaluation criteria (strong diagrams).

Strong Diagrams

Client-Server | gemini-1.5 | SRS_v1 | NoRAG
The diagram in Figure 4.3: DCC Experiment (ID = 67) satisfies nearly all the application's software requirements and adheres to the Client-Server architectural pattern. It clearly defines separate client and server packages, each containing the appropriate classes with sufficient attributes. On the client side, the diagram effectively integrates both the user interface and application logic, while the server side is dedicated to database management.

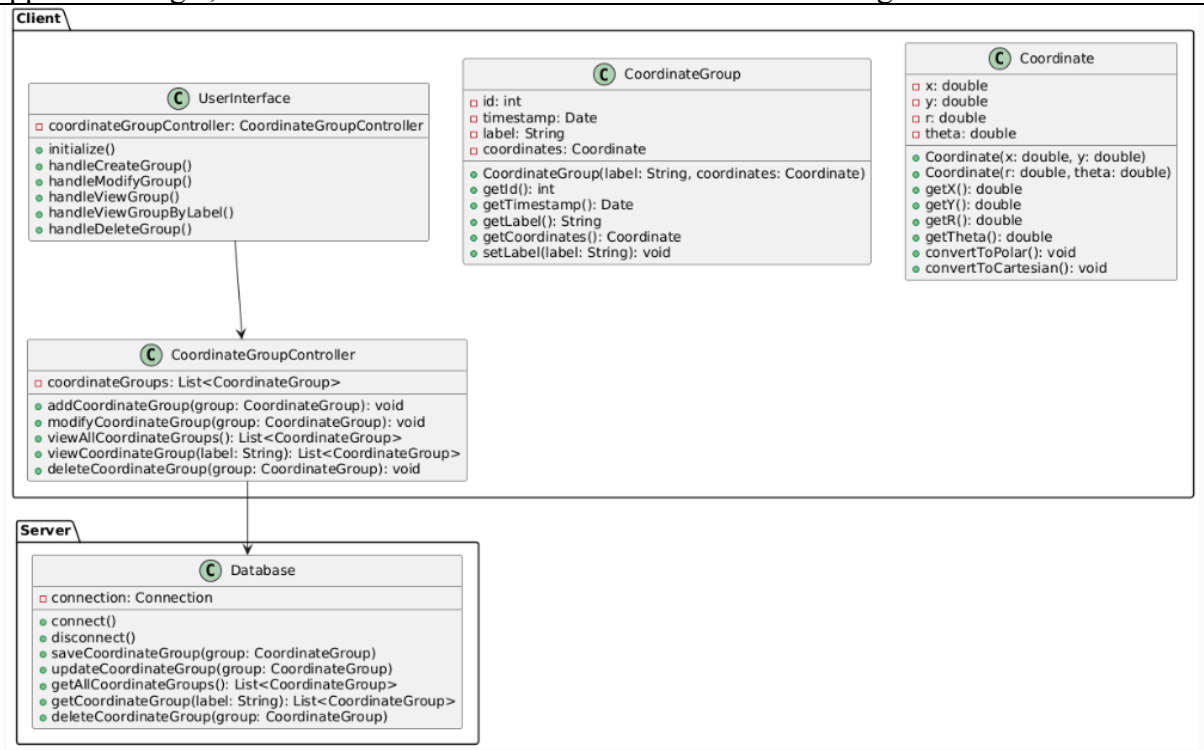


Figure 4.3: DCC Experiment (ID = 67)

Client-Server | gpt4o | SRS_v2 | NoRAG

Similarly, the class diagram in Figure 4.4 accurately defines two packages (client and server) each containing the appropriate classes for their respective responsibilities. The *client* package includes classes that handle CRUD operations, coordinate conversion and the graphical user interface, while the *server* package encapsulates the domain model and manages database operations. Although the diagram lacks some detail in the GUI implementation, it nonetheless presents a solid Client-Server design for the DCC application, satisfying nearly all software requirements.

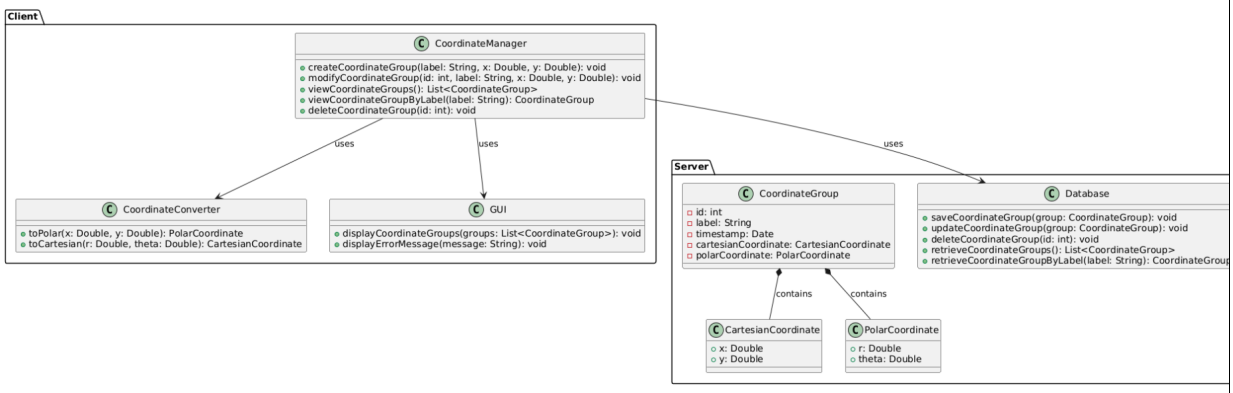


Figure 4.4: DCC Experiment (ID = 79)

Three-Tier | claudeSonnet3.5 | SRS_v1 | NoRAG

The diagram in Figure 4.5 correctly defines three packages: Presentation Layer, Business Layer and Data Layer. Each containing the appropriate classes for its designated responsibilities. The Presentation Layer manages user interface operations, while the Business Layer handles both CRUD functionality and core business logic, including communication with the database. The Data Layer contains the database and is responsible for ensuring data persistence. Collectively, these three layers implement the Three-Tier architecture effectively, addressing nearly all of the application's software requirements.

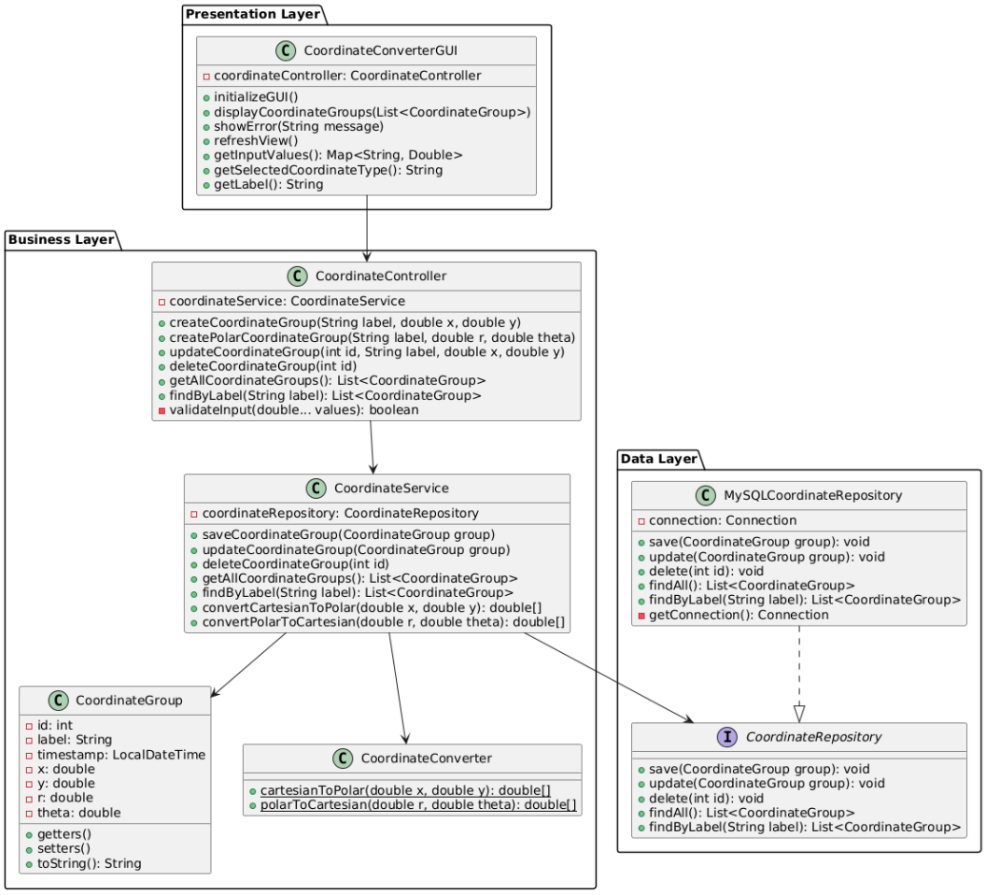


Figure 4.5: DCC Experiment (ID = 89)

Three-Tier | o1 | SRS_v1 | NoRAG

Likewise, the diagram in Figure 4.6 shows a clear structure with three packages: Presentation, Business and Data. The Presentation Layer takes care of the user interface. The Business Layer handles CRUD operations, business logic and connects to the database. The Data Layer stores the database and manages data storage. Overall, this design follows the Three-Tier architecture and meets most of the software requirements for the application.

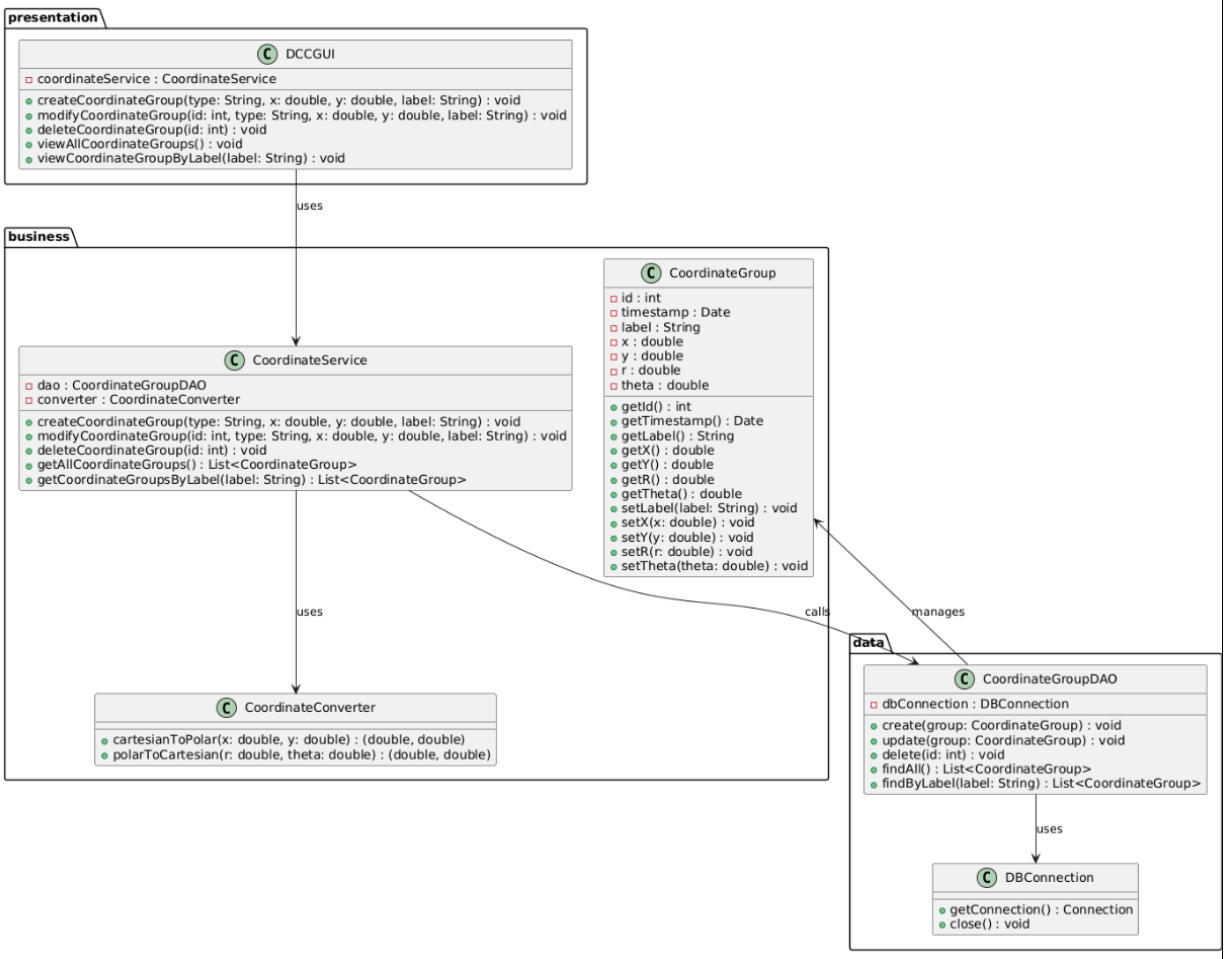


Figure 4.6: DCC Experiment (ID = 74)

MVC | deepseek-r1:70b | SRS_v2 | NoRAG

The class diagram in Figure 4.7 satisfies almost all software requirements and follows the MVC architecture. It includes four packages: View, Controller, Model and DAO. The View package contains the classes for the user interface, while the Controller package includes a class that handles CRUD operations. The Model package defines the coordinate group model and includes the business logic for coordinate conversion. The DAO package manages data access. The diagram also shows correct relationships between the classes.

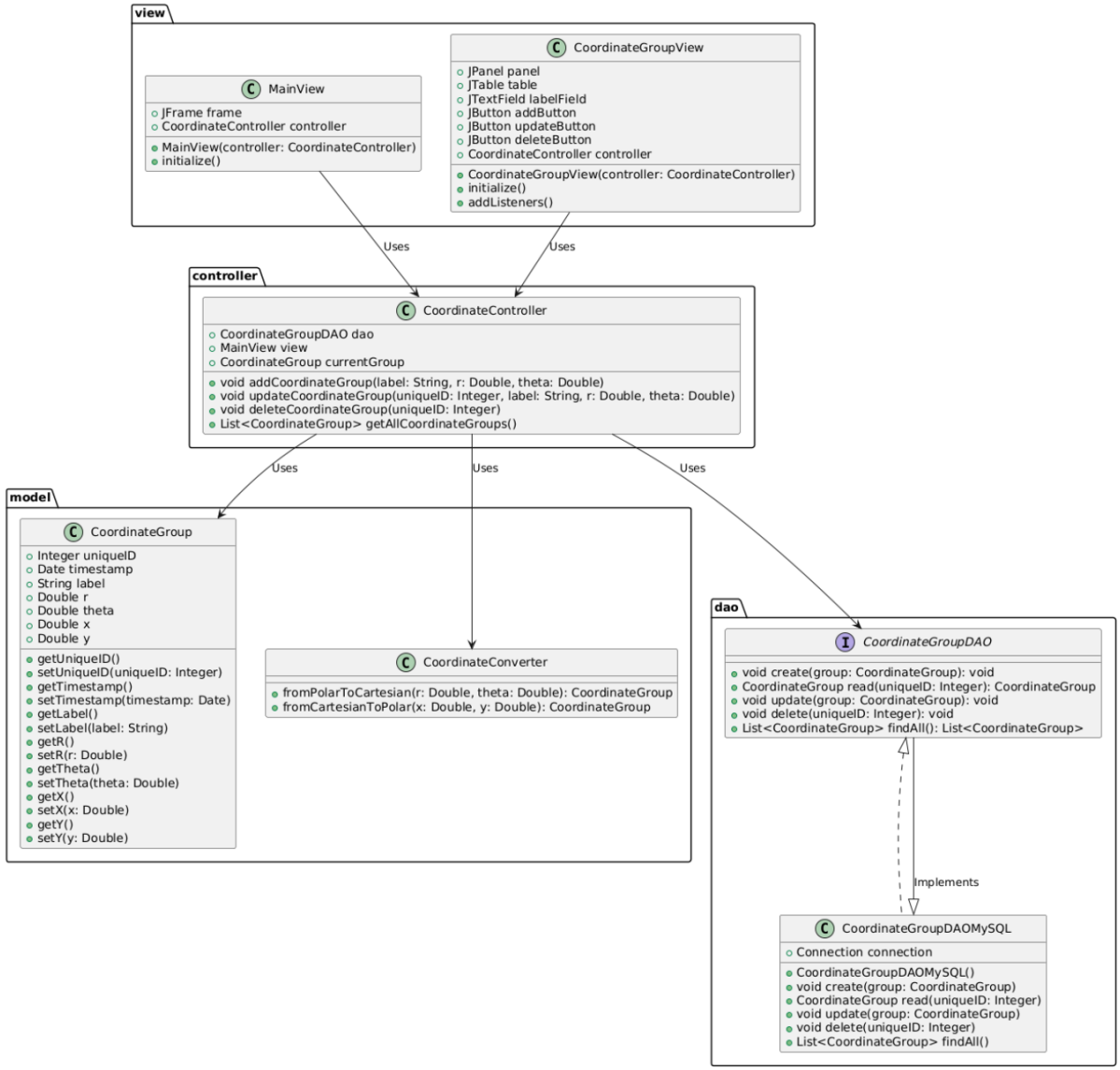


Figure 4.7: DCC Experiment (ID = 107)

MVC | o1 | SRS_v1 | NoRAG

Like the previous example, the diagram in Figure 4.8 aligns well with the MVC architecture and its principles. It includes clearly defined packages and classes, each with a very good level of detail.

Figure 4.8: DCC Experiment (ID = 75)

Weak Diagrams

Client-Server | command-r | SRS_v1 | RAG | ollama | nomic-embed-text | semantic
The class diagram in Figure 4.9 is unrelated to the DCC application and lacks any meaningful structure. It fails to define the core business operations and does not include a model for representing a coordinate group, making it unsuitable for the application's requirements.

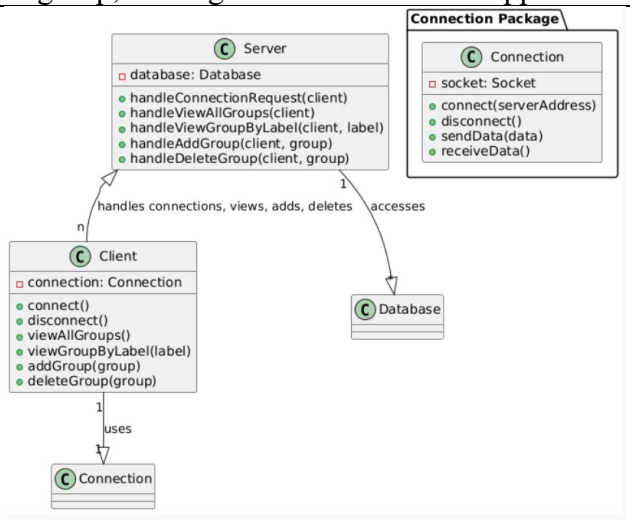


Figure 4.9: DCC Experiment (ID = 14)

Three-Tier | mixtral:8x7b | SRS_v1 | RAG | ollama | nomic-embed-text | semantic
The class diagram in Figure 4.10 does not conform to the principles of the Three-Tier architecture, as it lacks the essential separation into three distinct layers or packages: Presentation, Business and Data. Furthermore, the diagram fails to implement core business operations outlined in the software requirements, making it an incomplete and ineffective representation of the intended system design.

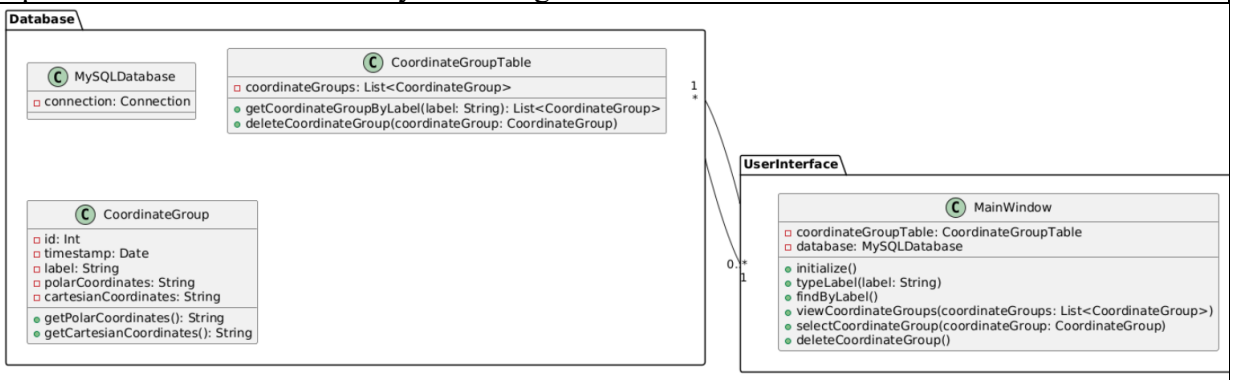
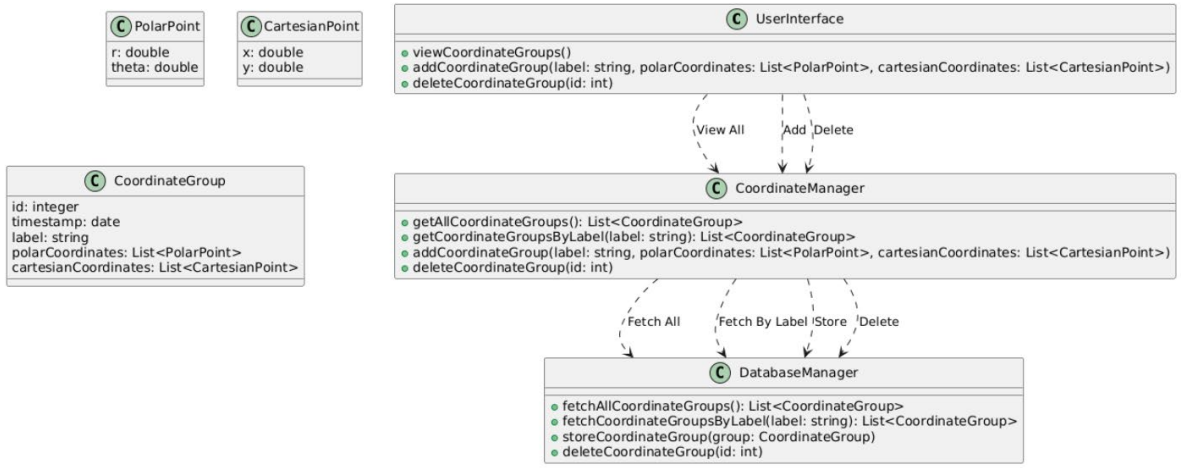


Figure 4.10: DCC Experiment (ID = 22)

MVC | command-r | SRS_v1 | NoRAG
While the diagram in Figure 4.11 includes a few relevant classes, such as CoordinateGroup and CoordinateManager, which partially reflect the software requirements, it fails entirely to follow any architectural pattern. Moreover, it does not utilize packages to organize the system's components, resulting in a disorganized and incoherent structure. Overall, the diagram represents a poorly generated output that lacks both structure and alignment with the application's design goals.
 *Figure 4.11: DCC Experiment (ID = 17)*

4.2.2 Evaluation Results

In this section, we present a series of charts based on evaluation data, highlighting key insights from the human evaluations of the class diagrams. All visualizations were created using the Highcharts library.

LLM Performance

The heatmap in Figure 4.12 illustrates the average evaluation scores of class diagrams generated by each LLM. It is important to note that the online models generated diagrams without Retrieval-Augmented Generation (RAG) and, as a result, produced significantly fewer diagrams compared to the local models.

Despite the smaller sample size, the online models appear to outperform their local counterparts. In particular, ClaudeSonnet3.5, deepseek-r1 (online version), o1 and o1-mini consistently achieved high scores across all evaluation criteria, positioning them as top performers.

That said, local models should not be dismissed. While some, such as llama3.1 and phi3:medium-128k, underperformed, others like deepseek-r1:70b and gemma2:27b delivered strong results with consistently high scores across all evaluation principles. These findings

suggest that certain local models remain highly competitive and are viable options for supporting software architecture generation tasks.

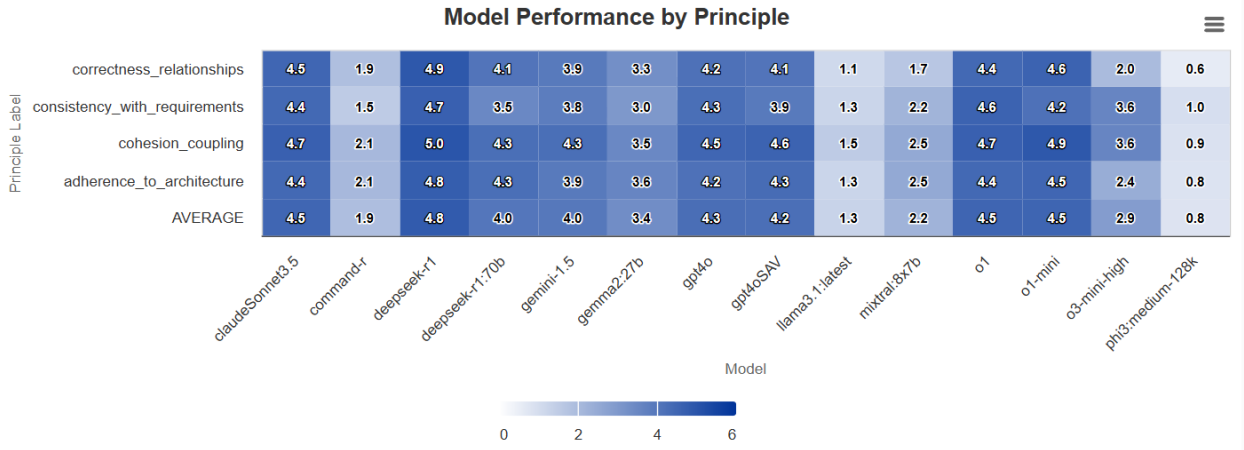


Figure 4.12: DCC Experiment - Model Performance

To explore the impact of different experimental parameters on the quality of the generated class diagrams, we begin by examining the role of Retrieval-Augmented Generation (RAG). The charts in Figure 4.13, Figure 4.14, Figure 4.15 display the average evaluation scores for each LLM, distinguishing between diagrams generated with RAG and those generated without it. The results reveal notable trends and offer promising insights, highlighting opportunities for future enhancements and optimizations in prompt design and model configuration.

These charts indicate that while some models such as deepseek-r1:70b and command-r, appear to benefit from the use of RAG, others show a decline in performance when RAG is applied. This unexpected result may be due to the models becoming overwhelmed or confused by the additional context introduced through RAG, especially when combined with the lengthy input from the Software Requirements Specification (SRS) document. Interestingly, in previous experiments where a simpler FR-NFR list was used instead of the full SRS, the same models demonstrated improved performance with RAG. This suggests that the effectiveness of RAG may depend heavily on the structure and complexity of the input context.

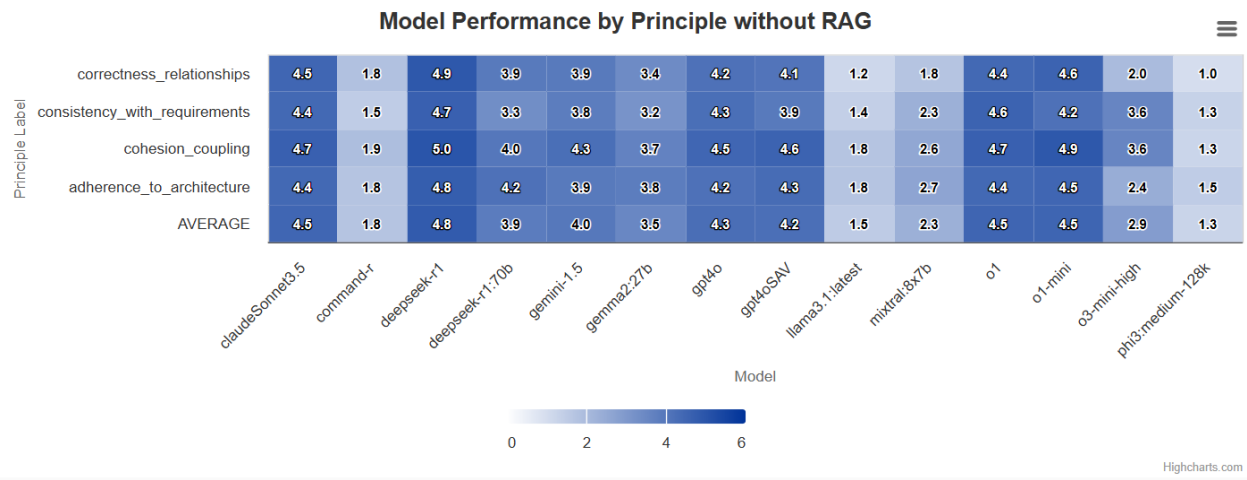


Figure 4.13: DCC Experiment - Model Performance No RAG

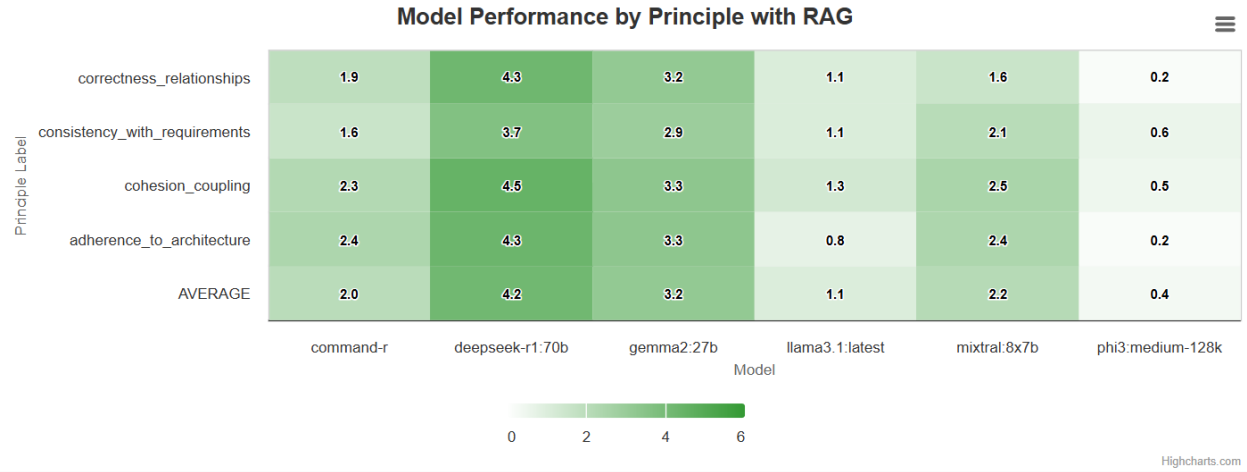


Figure 4.14: DCC Experiment - Model Performance with RAG

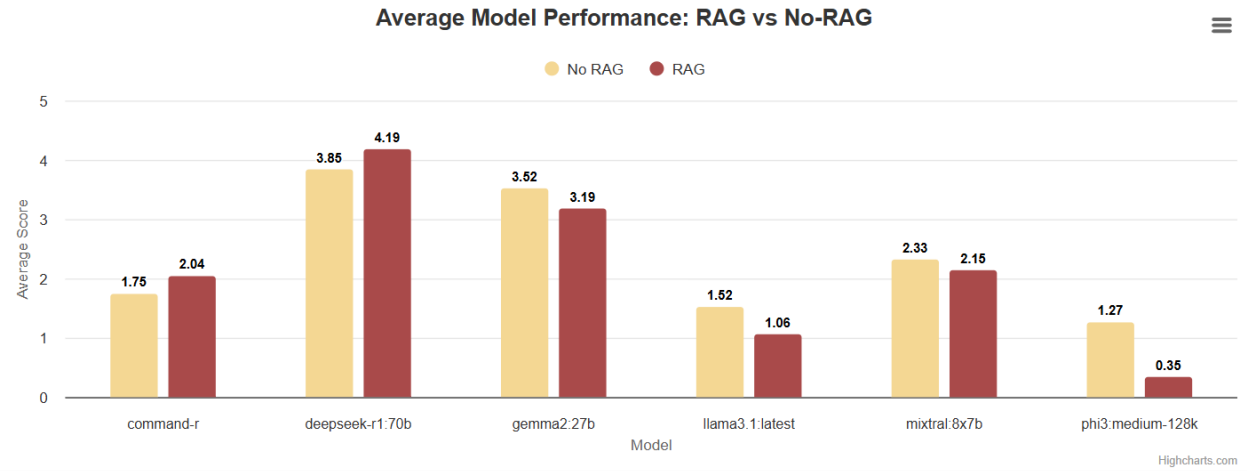


Figure 4.15: DCC Experiment - NoRAG vs RAG

Performance per Software Requirements Specification Document

To assess whether LLMs benefit from a detailed Software Requirements Specification (SRS) document compared to a more concise version, the diagram in Figure 4.16 compares the average performance scores across all models for each evaluation principle. The results suggest that the overall performance of the LLMs remains relatively consistent regardless of the length or depth of the SRS.

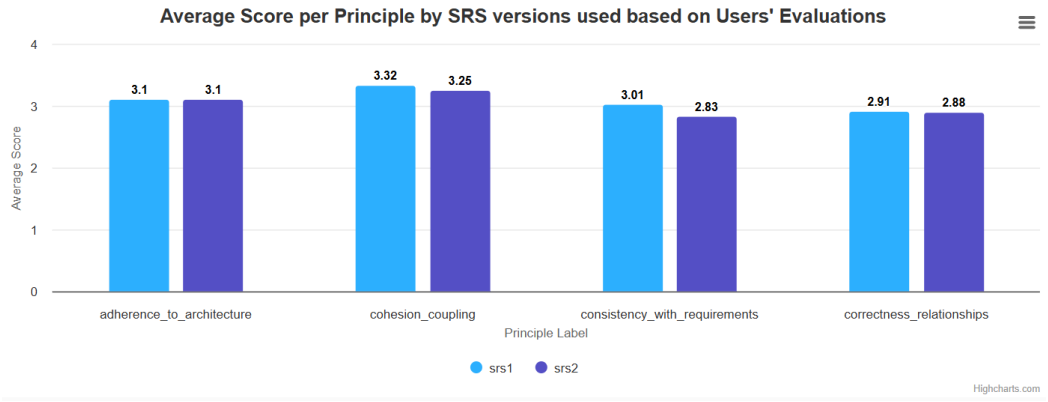


Figure 4.16: DCC Experiment - SRSv1 vs SRSv2

LLM Performance SRS vs FR-NFR

To investigate whether LLMs benefit more from detailed Software Requirements Specification (SRS) documents compared to simpler FR-NFR lists for describing a software application, this section presents a comparative analysis of their respective impacts on model performance without RAG, since the RAG average in the DCC experiment with FR-NFR includes multiple RAG methods that were not re-run in the SRS experiment. Figure 4.18 and Figure 4.17 display the average performance scores of the models using FR-NFR lists and SRS documents without RAG, respectively. Based on these results, we constructed the clustered bar chart below to provide a clear visual comparison of the two input formats.

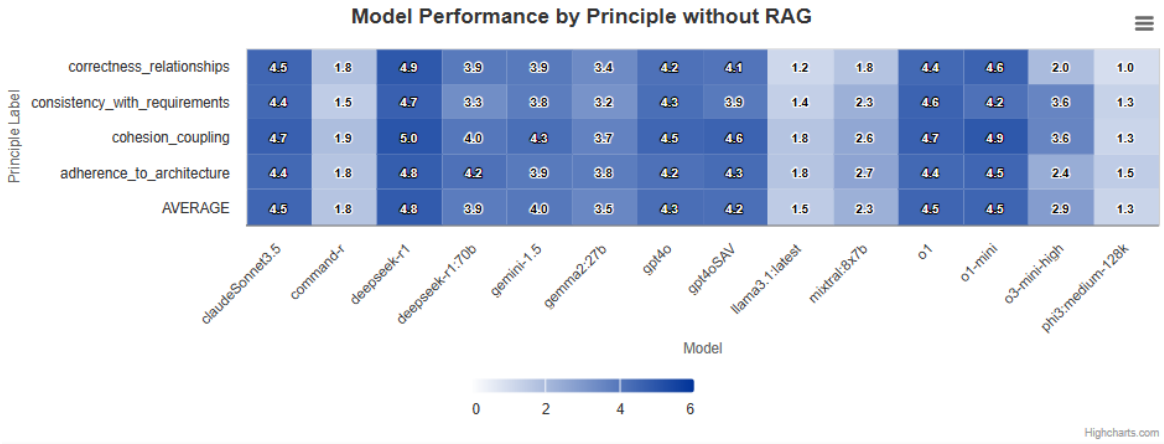


Figure 4.17: DCC Experiment - Model Performance SRS

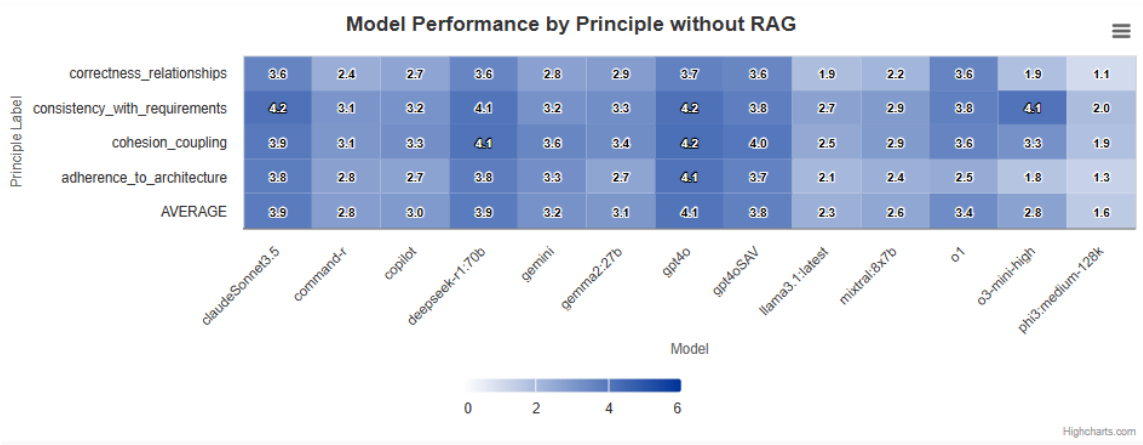


Figure 4.18: DCC Experiment - Model Performance - FR-NFR

The results reveal considerable variation, highlighting that not all LLMs benefit from more detailed software descriptions provided by SRS documents. Larger models with more parameters such as Claude Sonnet 3.5, Gemini-1.5, GPT-4o, GPT-4o SAV and o1 tend to perform better when using SRS input, generating class diagrams that receive higher average scores. In contrast, smaller local models like llama3.1 and mixtral:8x7b appear to struggle with the increased input length and complexity, performing significantly worse when given SRS documents compared to concise FR-NFR lists.

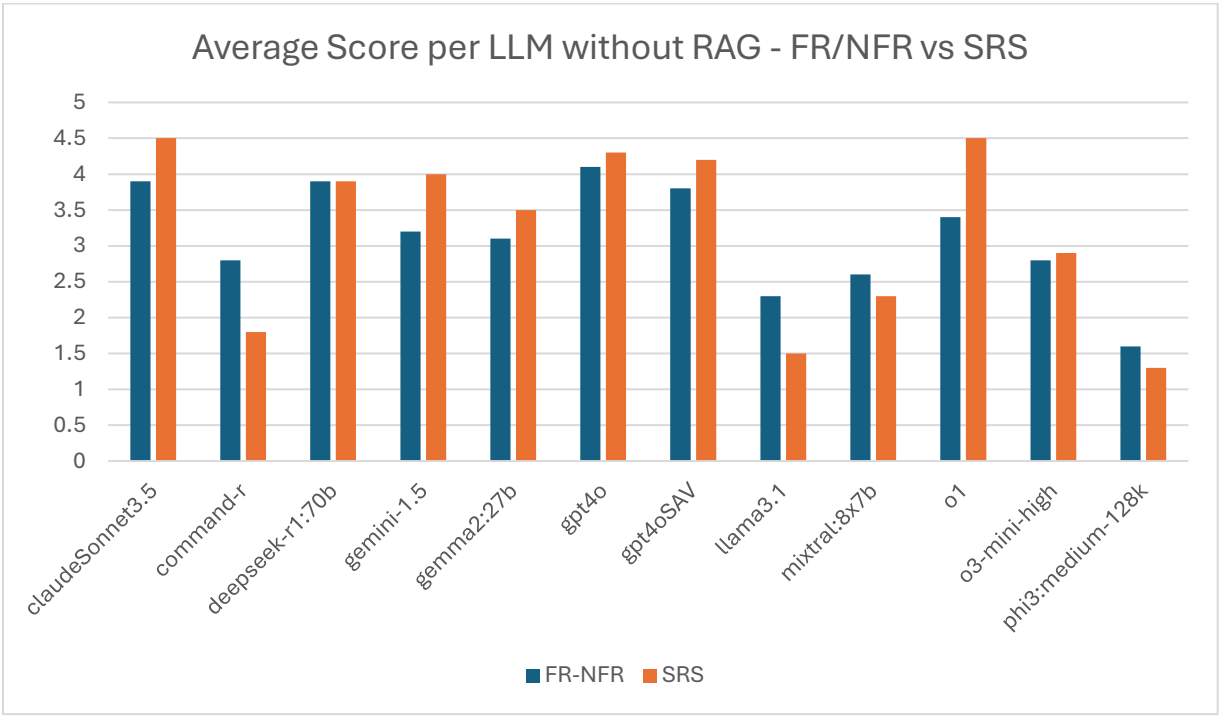


Figure 4.19: DCC Experiment - FR/NFR vs SRS

Performance per Architecture

Finally, we examine whether the choice of requested architecture influences the average scores of the generated class diagrams. The chart in Figure 4.20: DCC Experiment - Performance per Architecture displays the average scores grouped by architectural pattern.

The results in Figure 4.20 show that diagrams generated for the Client-Server architecture received generally the lowest average scores. In contrast, Three-Tier and Model-View-Controller (MVC) architectures performed more closely, with Three-Tier slightly outperforming MVC.

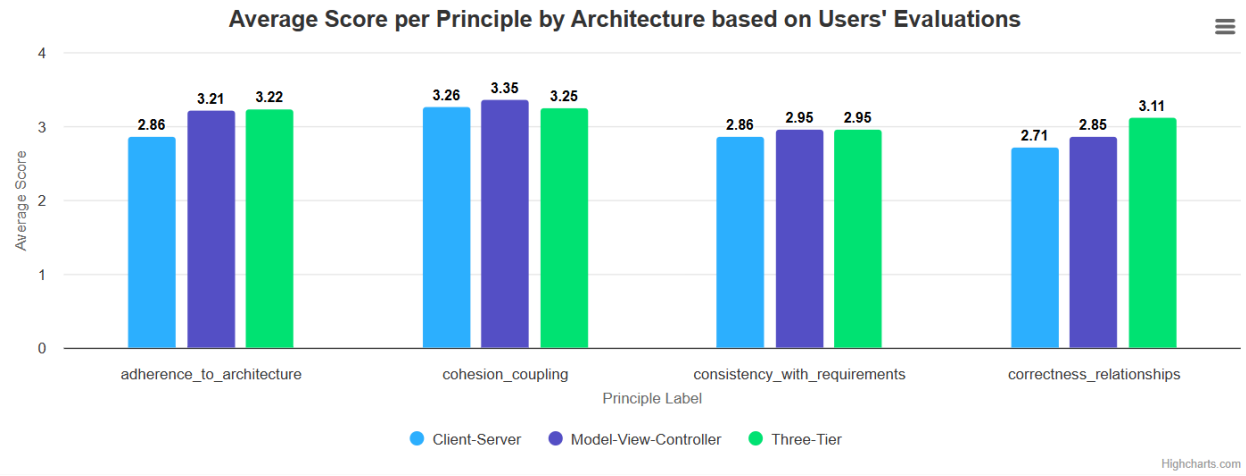


Figure 4.20: DCC Experiment - Performance per Architecture

4.2.3 Results Discussion

The analysis of our results reveals several noteworthy patterns regarding the factors that influence how well LLMs generate class diagrams, in relation to the experiment context and architectural patterns used. While human evaluation scores are subjective, the overall trends align well with the evaluators' qualitative impressions. A larger-scale expert review would further validate these findings, but even with the current data, a few clear takeaways emerge.

- **Larger online models generally performed better than smaller local ones**, producing more complete and well-structured diagrams across the board. This was expected, given their higher capacity and training scale.
- **The use of RAG led to mixed outcomes** when paired with the longer input from the full Software Requirements Specification (SRS). Some models clearly benefited from the additional context, while others seemed to struggle, possibly due to input overload or difficulty in focusing on the relevant parts.
- **When comparing detailed versus concise versions of the SRS**, model performance remained relatively consistent, suggesting that more verbose descriptions don't necessarily improve diagram quality.

- **Performance comparison between SRS documents and FR-NFR lists**. Larger models tended to handle the full SRS documents better and generated higher-quality diagrams, while smaller models often performed worse with the longer input and showed better results when working with the more focused FR-NFR lists.
- **The requested software architecture also played a role in performance**. Diagrams generated for the Client-Server architecture consistently received lower scores, indicating that this pattern may be more challenging for LLMs to interpret and model correctly. In contrast, the Three-Tier and MVC architectures were handled more effectively, with Three-Tier slightly outperforming MVC.

4.3 MyCharts Experiment

4.3.1 Typical Cases

In this section, we present a selection of UML class diagrams generated by the LLMs for the *MyCharts* experiment. As anticipated, the quality of the diagrams varied significantly, from incomplete and unstructured representations with little relevance to the *MyCharts* application (weak diagrams), to well-structured and highly accurate class diagrams that performed exceptionally across all seven evaluation criteria (strong diagrams).

Strong Diagrams

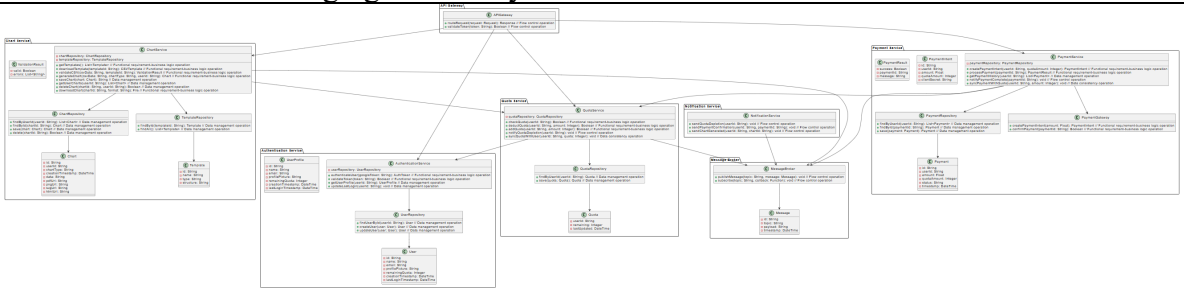
Microservices | claudeSonnet3.7 | fr | nfr | NoRAG
The diagram in Figure 4.21 illustrates a well-structured Microservices architecture for the *MyCharts* application. Responsibilities are logically distributed across the microservices and collectively satisfy all the application's requirements. The architecture adheres to design principles, including low coupling and high cohesion, because each microservice has a degree of independence and can be deployed atomically, yet several services collaborate to complete end-to-end use cases. Each microservice manages its own dedicated database or repository, supporting the principle of decentralized data management and includes operations for eventual consistency where necessary. Communication across services adheres to standard Microservices patterns, utilizing an API Gateway for request routing and a message broker to handle inter-service messaging effectively.


Figure 4.21: MyCharts Experiment – FR/NFR (ID =16)

Microservices | grok3 | SRS | NoRAG

Similarly, the diagram in Figure 4.22 presents a well-structured Microservices architecture. Responsibilities are logically distributed among the microservices and the design meets nearly all application requirements. Like the previous example, it demonstrates low coupling and high cohesion, with each microservice managing its own data and including mechanisms to maintain consistency. In terms of communication, this diagram employs an API Gateway for routing client requests and adopts a choreography-based approach for inter-service interactions, aligning well with modern Microservices best practices.

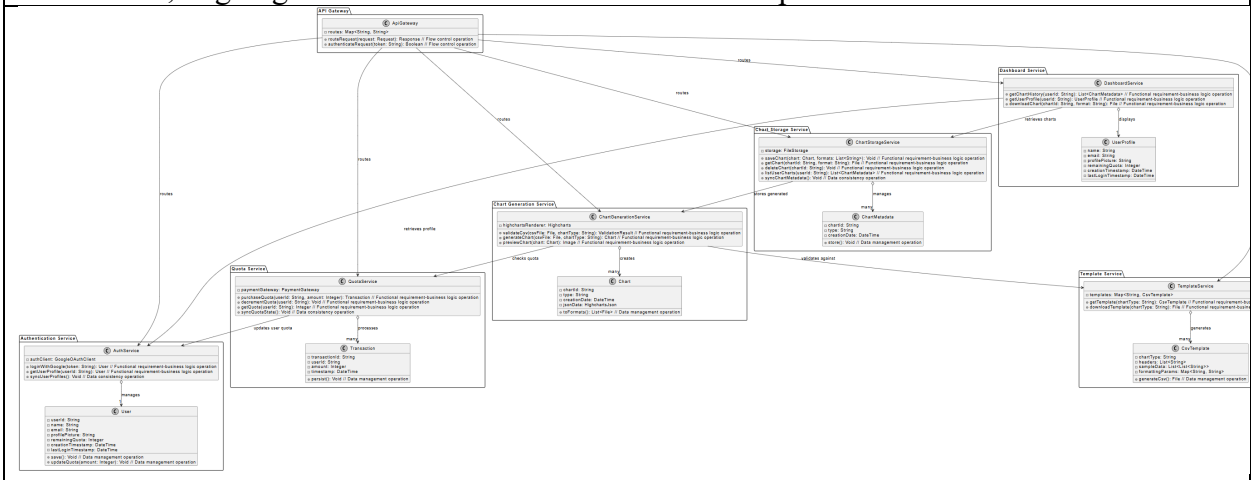


Figure 4.22: MyCharts Experiment - SRS (ID = 21)

Weak Diagrams

Microservices | mistral | SRS | RAG | ollama | nomic-embed-text | semantic

The diagram in Figure 4.23 illustrates an example of a weak response generated by the LLM, reflecting significant issues in both understanding the software requirements and applying the principles of Microservices architecture. Key functionalities are missing, services are isolated with no interconnections and there is a complete absence of databases or repositories. As a result, the diagram appears incomplete and lacks a coherent structure, failing to meet the basic expectations for a Microservices-based design.

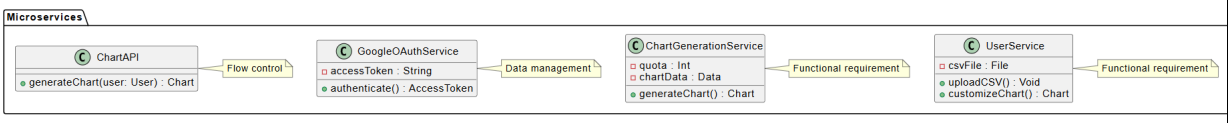


Figure 4.23: MyCharts Experiment - SRS (ID = 8)

Microservices | mixtral:8x22b | fr | nfr | RAG | ollama | nomic-embed-text | semantic
The class diagram in Figure 4.24 fails to follow any fundamental principles of established software architectures and does not satisfy any of the defined software requirements. The structure lacks classes necessary for the representation of the intended system.

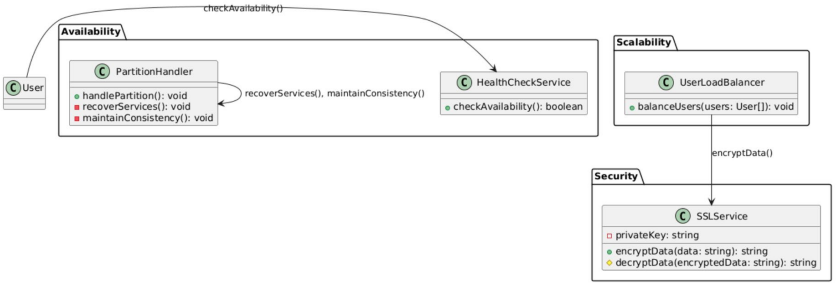



Figure 4.24: MyCharts - FR/NFR (ID = 14)

Noteworthy Diagrams

The diagrams presented in this section may not achieve high scores across all evaluation criteria, as some requirements are either overlooked or the designs do not fully align with Microservices architecture principles. However, they still represent surprisingly well-structured responses from local models that were initially expected to underperform, highlighting their potential in generating coherent architectural designs.

Microservices | gemma2:27b | fr | nfr | RAG | ollama | nomic-embed-text | semantic
The diagram in Figure 4.25, despite missing several key elements, demonstrates clear potential. It lacks support for certain software requirements, such as template downloading and chart validation and shows limited detail in the implementation. Additionally, it does not distribute functional responsibilities across a wide range of microservices, relying instead on just a few. However, the response reflects a general understanding of both the application and Microservices architecture. Notably, it incorporates an API Gateway that handles request routing and also functions as an orchestrator. Overall, this is a reasonably solid output from a local LLM and could serve as a strong foundation for further refinement.

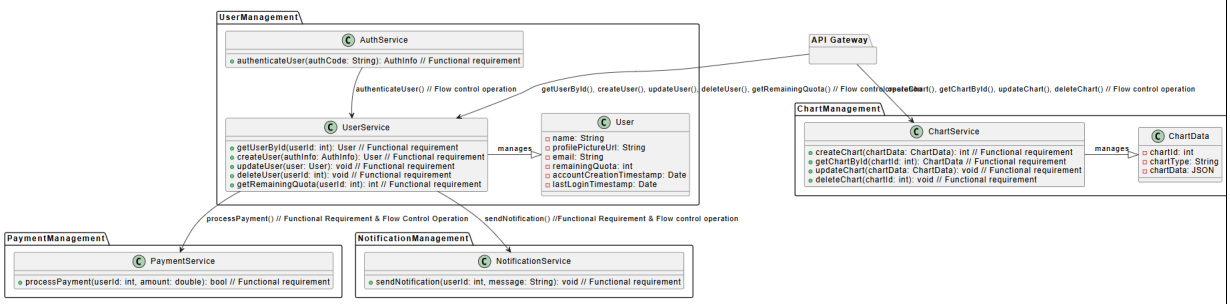



Figure 4.25: MyCharts Experiment - FR/NFR (ID = 12)

Microservices | deepseek-r1:70b | fr | nfr | RAG | ollama | nomic-embed-text | semantic
Similarly, the diagram Figure 4.26, while lacking some essential characteristics of a Microservices architecture, shows notable potential. One key omission is the absence of dedicated databases for each microservice. Despite this, the diagram demonstrates a solid grasp of Microservices principles by logically distributing functional responsibilities and addressing nearly all software requirements. It also features an API Gateway that manages request routing and acts as an orchestrator. Again, this is a promising response from a local LLM and could serve as a strong foundation.

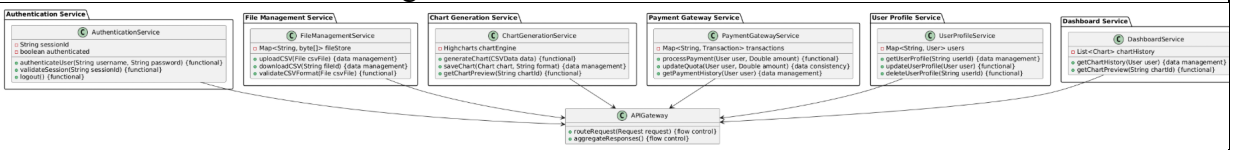


Figure 4.26: MyCharts Experiment - FR/NFR (ID = 6)

4.3.2 Evaluation Results

In this section, we present a series of charts based on evaluation data, highlighting key insights from the human evaluations of the class diagrams. All visualizations were created using the Highcharts library.

It is important to note that in this experiment, we intentionally limited the number of generated diagrams to 46, significantly fewer than in previous studies. This decision was made to allow for a more in-depth evaluation based on seven detailed criteria, along with performance analysis using manually calculated objective metrics. As a result, we worked with a smaller set of scenarios. These constraints should be kept in mind when interpreting the average performance of the LLMs, as the findings are based on a relatively small sample size.

LLM Performance

The heatmaps in Figure 4.27, Figure 4.28 display the average evaluation scores of class diagrams produced by each LLM in FR-NFR and SRS experiments respectively. It is important to highlight that online models generated diagrams without the use of Retrieval-Augmented Generation (RAG). In contrast, local models produced three versions of each diagram: one without RAG, one using the rag1 file as supplemental material and another using the rag2 file.

Across both FR-NFR and SRS input scenarios, online models such as ClaudeSonnet3.7, o1, o3-mini-high, deepseek-r1 and gemini-2.0 consistently outperformed local models, as expected. However, certain local models demonstrated strong performance in the FR-NFR cases, with gemma2:27b and deepseek-r1:70b achieving scores that were comparable with their online competitors.

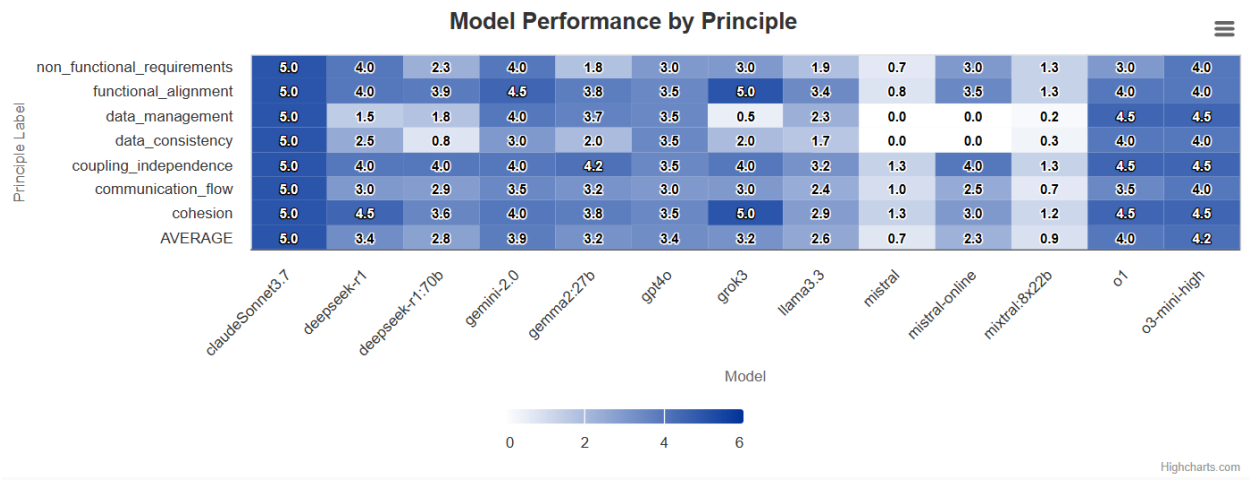


Figure 4.27: MyCharts Experiment - Model Performance (FR/NFR)

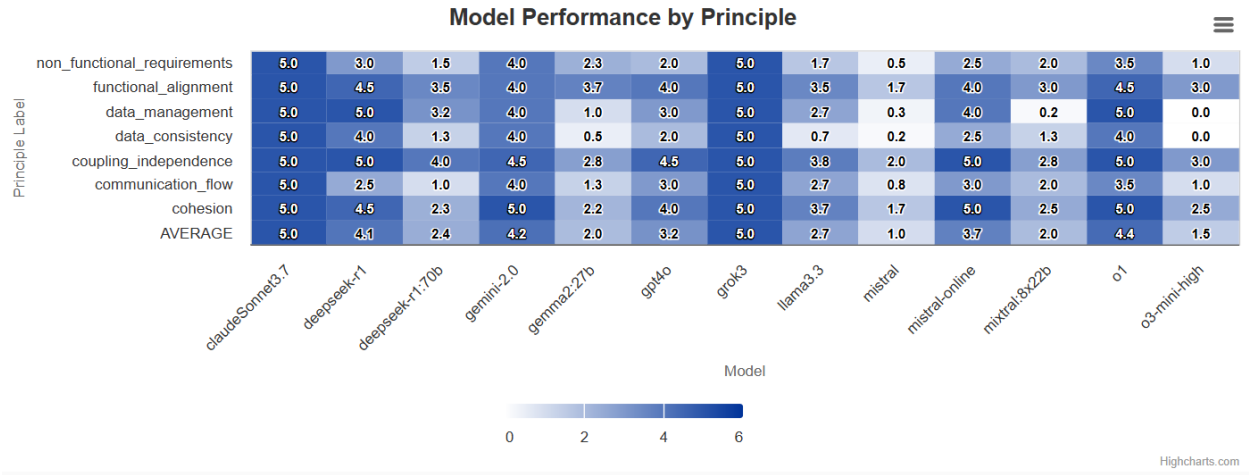


Figure 4.28: MyCharts Experiment - Model Performance (SRS)

The chart in Figure 4.29 compares the average performance of LLMs across all evaluation criteria when using FR/NFR lists versus a full SRS document as input. A consistent trend persists, larger LLMs with more parameters tend to perform better when given the more detailed and extensive input provided by the SRS document, as opposed to the concise FR/NFR lists. On the other hand, most local LLMs struggle with the increased input length, often becoming overwhelmed and underperforming. A notable exception is mixtral:8x22b, a local model with 141 billion parameters, which appears to handle the richer input effectively and benefits from the additional context provided by the SRS due to its parameter count.

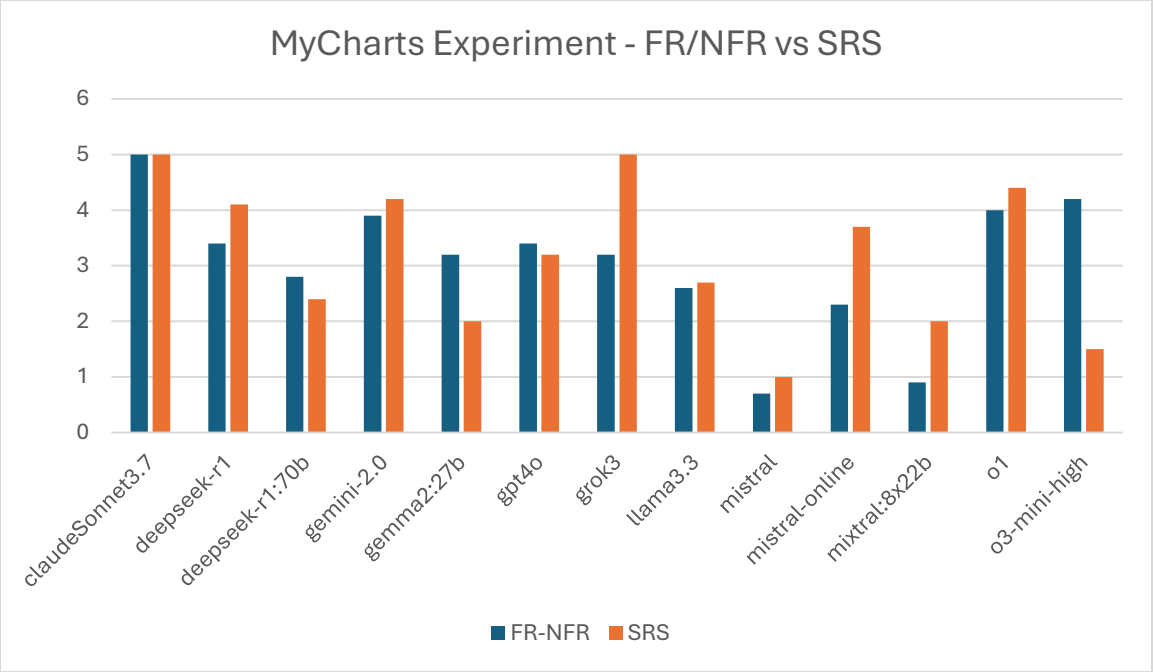


Figure 4.29: MyCharts Experiment - FR/NFR vs SRS

NoRAG vs RAG

In this section, we compare the performance of class diagrams generated with and without Retrieval-Augmented Generation (RAG) across both FR/NFR and SRS input formats. The goal of this analysis is to examine whether RAG methods provide measurable benefits when applied to a more complex application like *MyCharts* and a more demanding architectural pattern such as Microservices.

FR/NFR Input:

The charts in Figure 4.30, Figure 4.31 indicate that, when combined with FR/NFR input, certain local models such as deepseek-r1:70b and gemma2:27b, demonstrate improved performance with RAG. In contrast, other models like llama3.3, mistral and mixtral:8x22b tend to underperform when RAG is applied.

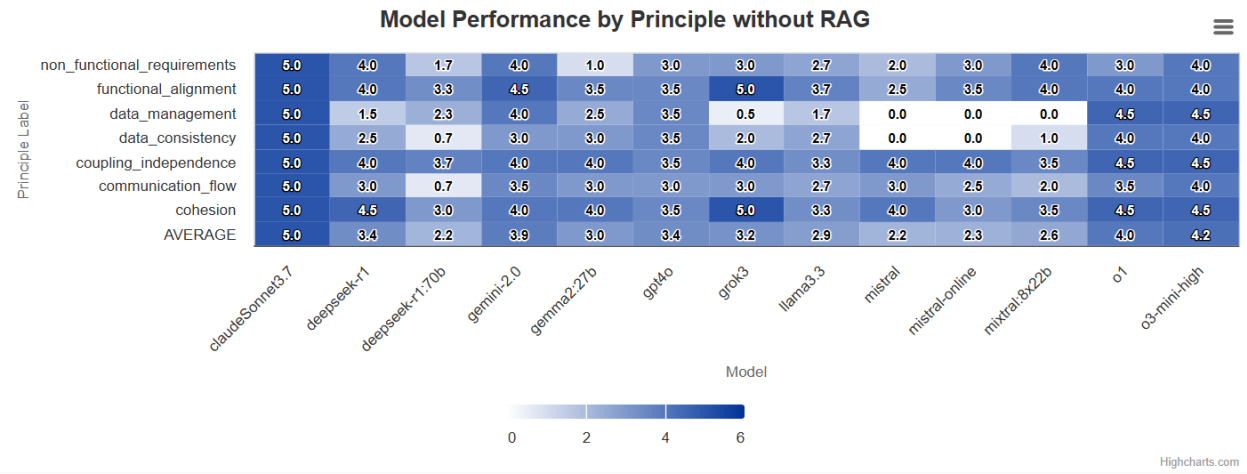


Figure 4.30: MyCharts Experiment - Model Performance without RAG (FR/NFR)

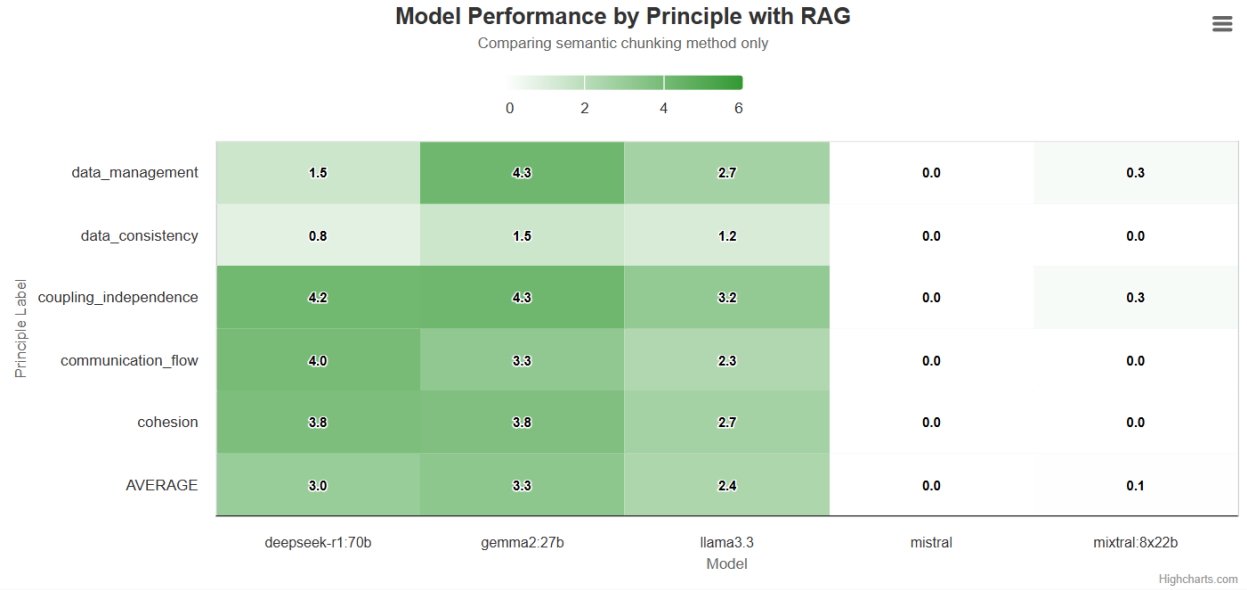


Figure 4.31: MyCharts Experiment - Model Performance with RAG (FR/NFR)

The chart in Figure 4.32 illustrates the impact of RAG methods on the performance scores of each model, while also highlighting the comparative effectiveness of the two RAG files introduced in Chapter 3.3.7. Among the models that benefited from RAG, the rag2 file, which is designed as a more concise and practical explanation of the Microservices architecture, consistently led to better performance. This suggests that targeted, streamlined supplemental material can be more effective than lengthier or more general alternatives in guiding LLMs.

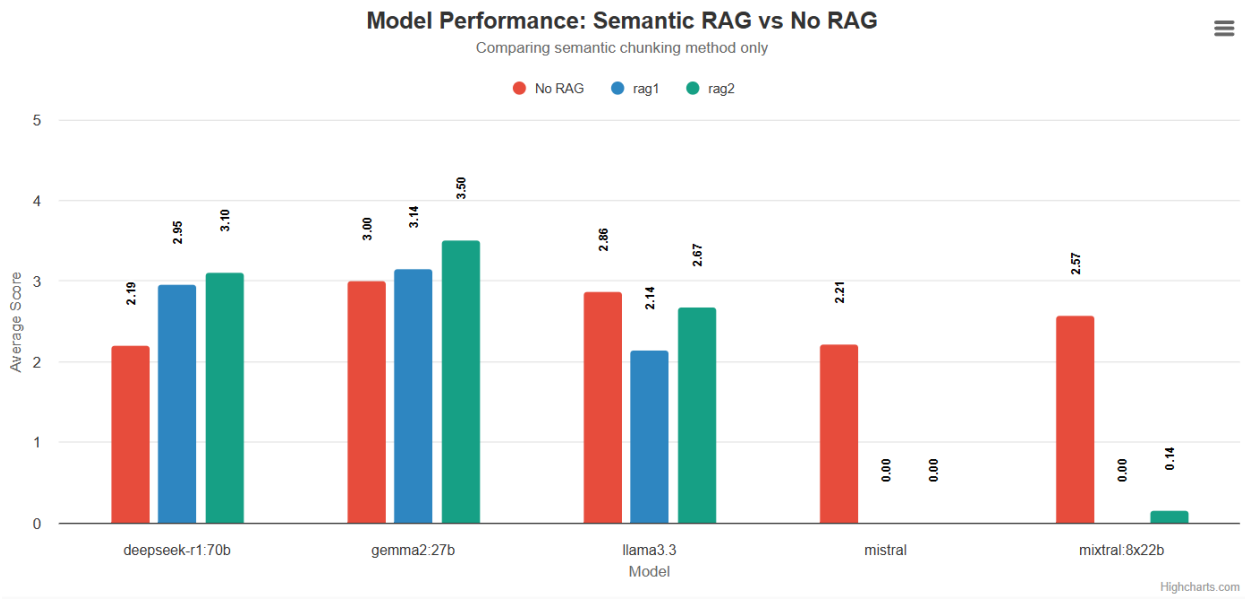


Figure 4.32: MyCharts Experiment - NoRAG vs RAG (FR/NFR)

SRS Input:

The charts in Figure 4.33, Figure 4.34 indicate that, when combined with SRS input, nearly all local models, except mistral, demonstrate improved performance with RAG on average.

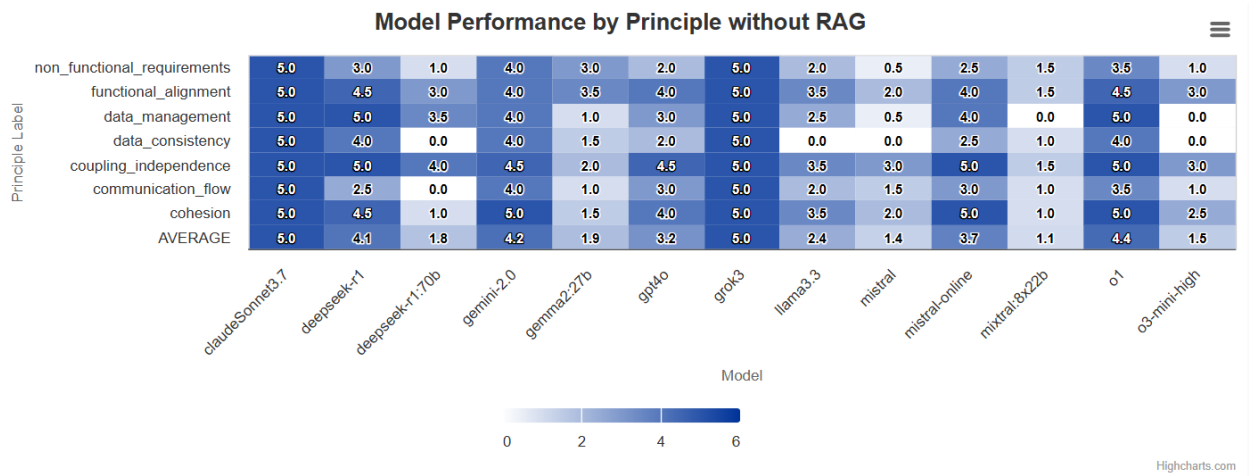


Figure 4.33: MyCharts Experiment - Model Performance without RAG (SRS)

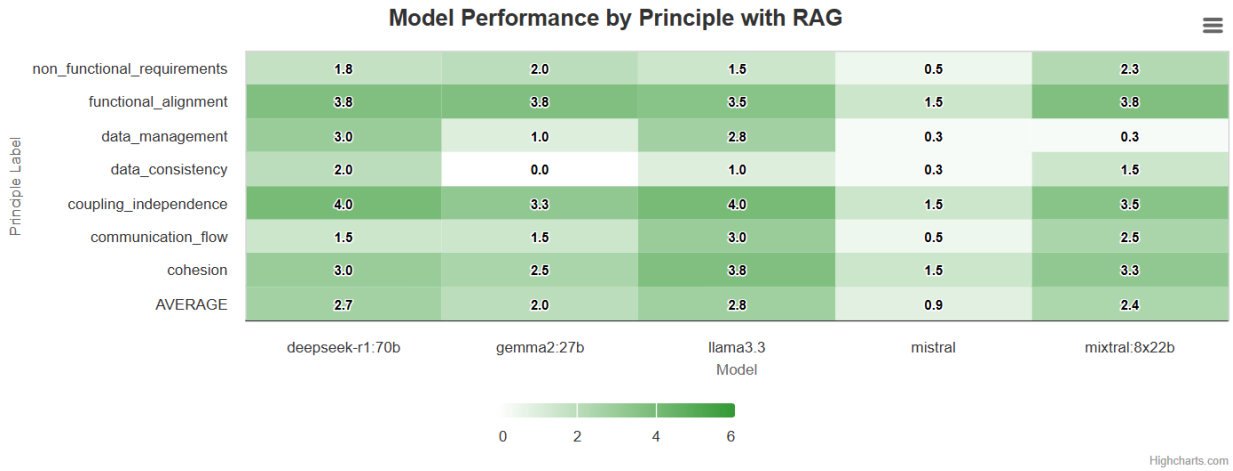


Figure 4.34: MyCharts Experiment - Model Performance with RAG (SRS)

The chart in Figure 4.35 showcases the impact of RAG methods on the performance scores of each model, while also highlighting the comparative effectiveness of the two RAG files introduced in Chapter 3.3.7. Across all models, the rag2 file, which offers a concise and practical explanation of the Microservices architecture, consistently resulted in better performance. This finding suggests that well-targeted and streamlined supplemental material can be more effective in guiding LLMs than longer or more generalized documents.

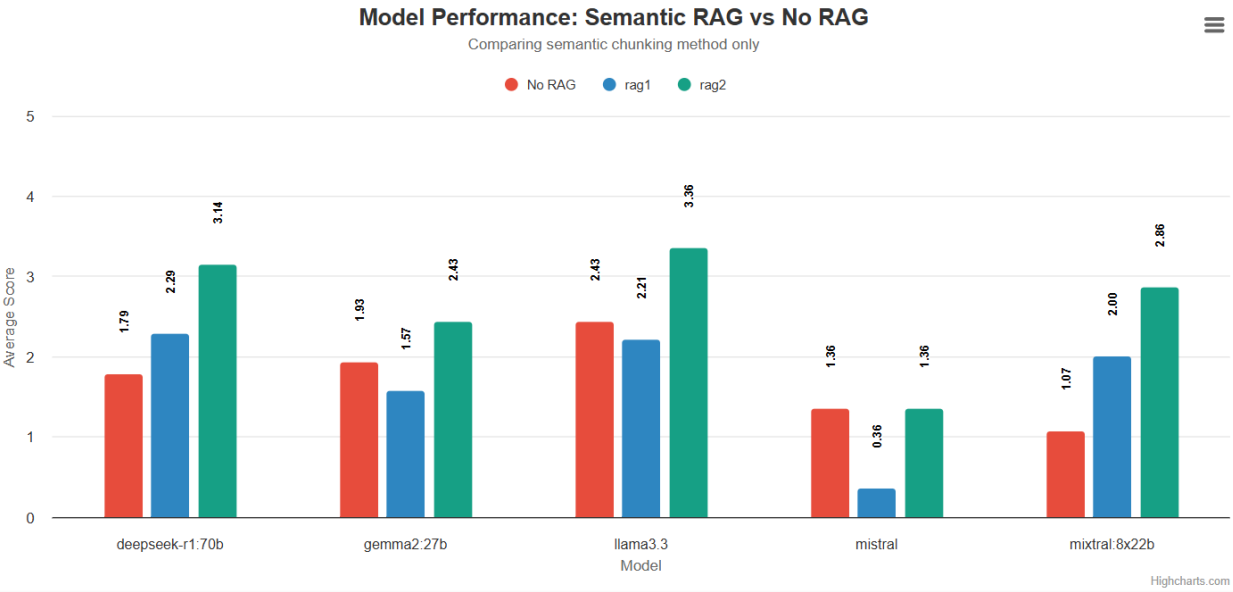


Figure 4.35: MyCharts Experiment - NoRAG vs RAG (SRS)

4.3.3 Metric Performance

To introduce a more objective dimension to the evaluation process for the Microservices architecture, this experiment incorporates the metrics outlined in Chapter 3.3.4. As discussed earlier, there is a conceptual alignment between these metrics and the subjective evaluation principles. In this section, we examine whether this alignment is reflected in practice through experimental results.

Following the *MyCharts* experiment pipeline stated in Figure 3.8, after generating the class diagrams, we manually calculated the defined metrics for each one. To enable a fair comparison, we normalized both the metric values and the human evaluation scores to a scale from 0 to 1. The human evaluations were normalized individually for each evaluation principle to maintain consistency across different criteria.

As outlined in Chapter 3.3.4, the quality of a generated class diagram can be interpreted through the following metric patterns: SI, DOC and SST should ideally be close to 1 indicating stateless microservices that manage their own data and have operations achieving eventual data consistency. SC should reflect a high number of services per use case, signaling appropriate service cohesion. In contrast, lower values are desirable for SIC, AIS and ADS, as they indicate reduced inter-service coupling and redundancy, for this reason, these metrics have been inverted in the following charts to align with the convention that higher values indicate better-quality diagrams.

The charts in Figure 4.36, Figure 4.37, Figure 4.38, Figure 4.39, Figure 4.40 display on the horizontal axis the ID of the class diagrams generated, where 1-23 refer to FR/NFR scenarios and 24-46 refer to corresponding 1-23 of SRS scenarios and the vertical axis represents the score either of the metric of the evaluation. These charts provide a visual representation of the conceptual correlation between some evaluation principles with some of the metrics.

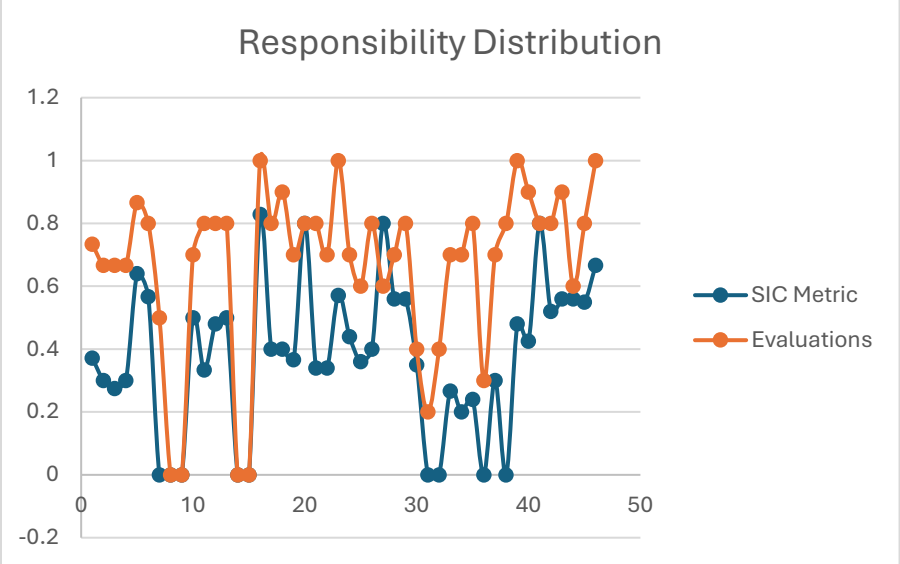


Figure 4.36: MyCharts Experiment - Metric Correlation - Responsibility Distribution

Figure 4.37: MyCharts Experiment - Metric Correlation - Data Management

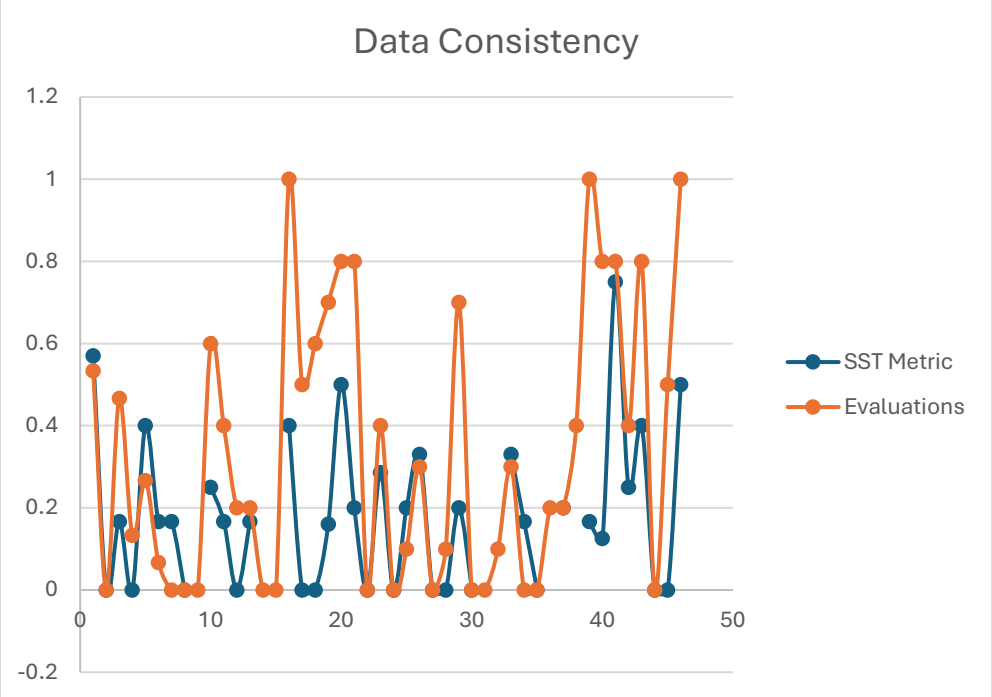


Figure 4.38: MyCharts Experiment - Metric Correlation - Data Consistency

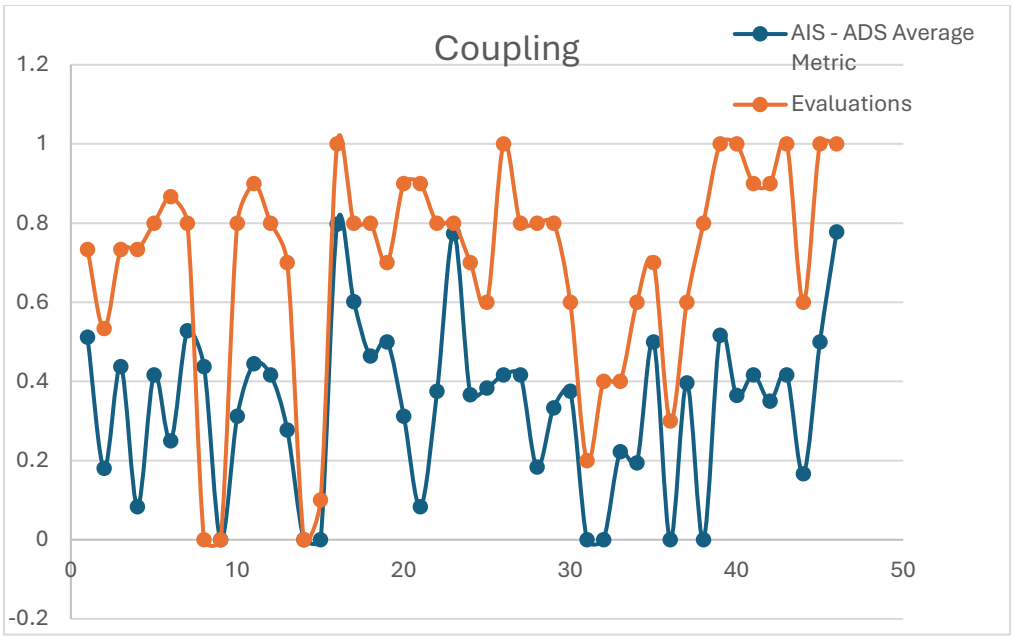


Figure 4.39: MyCharts Experiment - Metric Correlation – Coupling

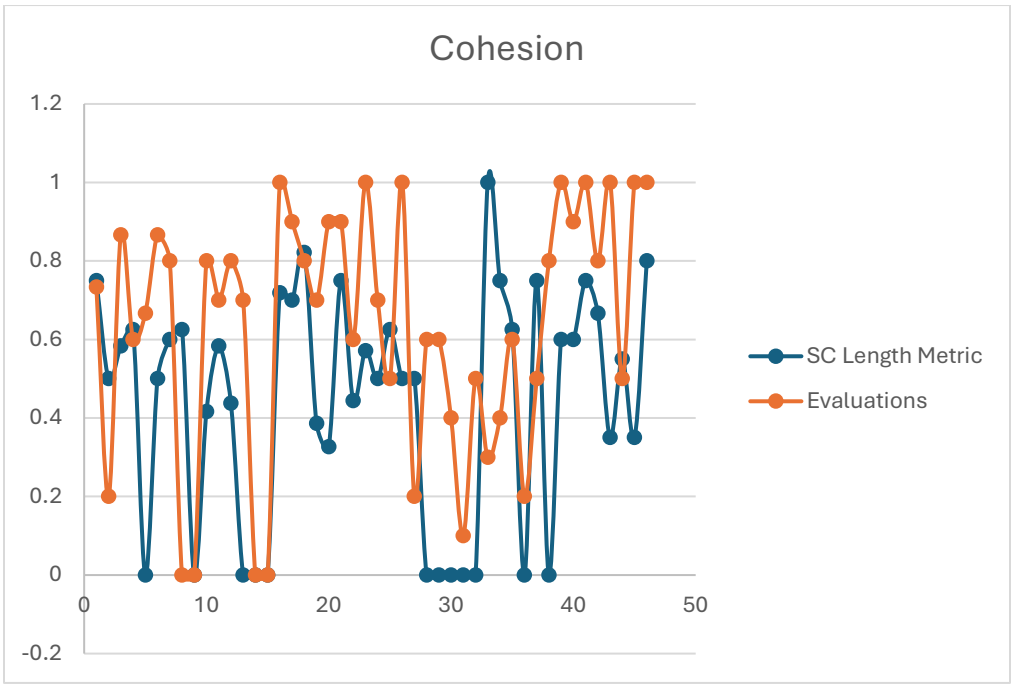


Figure 4.40: MyCharts Experiment - Metric Correlation - Cohesion

The charts illustrate that for certain evaluation principles such as Data Management, there is a clear and observable correlation between the subjective evaluations and the corresponding metrics. However, for other principles like Cohesion, the relationship is less evident. This suggests that further investigation is needed, including the exploration of additional or alternative metrics, to more accurately and objectively capture the concept of cohesion in class diagrams.

4.3.4 Metric Hallucination

To assess the extent to which LLMs exhibit hallucination in metric generation, i.e., calculating inaccurate metric values, we compared the manually calculated metrics for each class diagram with the corresponding metrics automatically generated by the LLMs. The chart in Figure 4.41 illustrates the difference between these two sets of values, effectively calculating the degree of hallucination per LLM and per metric.

As seen from the chart, it seems that all LLMs hallucinate to some extent, but the degree of hallucination varies significantly between models and across different metrics. Notably, models like mistral, o1 and o3-mini-high show particularly high hallucination in certain metrics, especially the SIC (Service Interface Count) and AIS (Absolute Importance of Service Average) metrics, with values approaching 1. This suggests a substantial mismatch between the automatically generated and the manually verified values. On the other hand, models such as claudeSonnet3.7, gemini-2.0 and mixtral:8x22b generally exhibit lower hallucination rates across most metrics, indicating a higher level of accuracy in self-assessed metric reporting. Interestingly, even strong performers like grok3 show high hallucination levels, especially in the SIC and DOC metrics, highlighting that performance in diagram generation does not always correlate with accurate self-evaluation. Overall, the variation in hallucination emphasizes the need for cautious interpretation of LLM-generated metrics and the importance of external verification when evaluating architectural outputs.

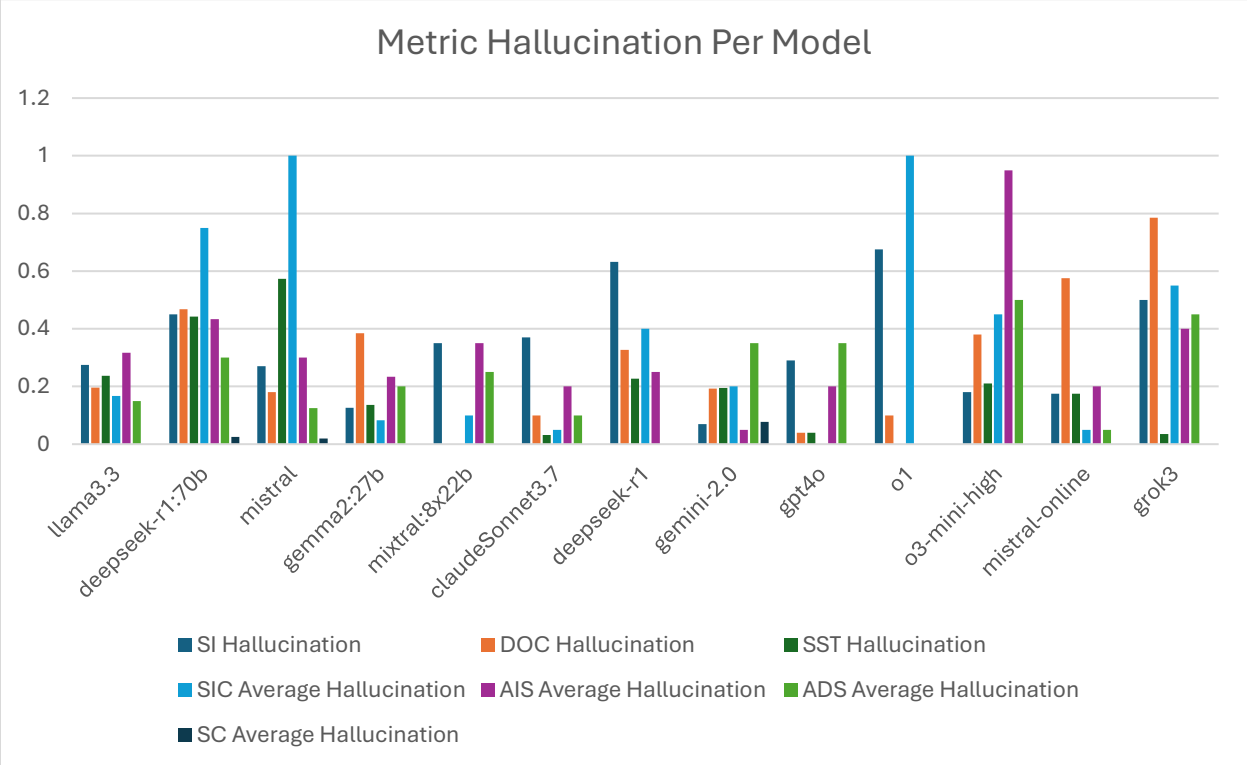


Figure 4.41: MyCharts Experiment - Metric Hallucination

4.3.5 Results Discussion

Our analysis highlights some clear patterns in how different factors impact the ability of large language models (LLMs) to generate class diagrams, especially for complex applications like *MyCharts* and when using more demanding architectures like Microservices. While human evaluations are subjective, the overall trends match well with the evaluators' impressions. A larger-scale experiment would help confirm these results, but even from our current data, several takeaways stand out:

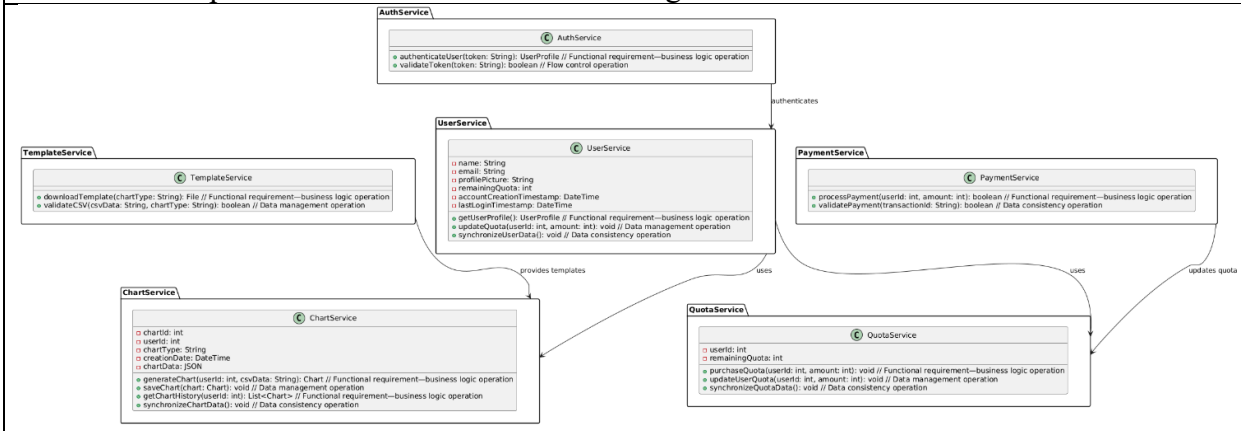
- **Bigger models generally do better:** Larger online LLMs outperformed smaller local ones, creating more complete and well-structured diagrams overall. This isn't too surprising, given their greater capacity and broader training.
- **LLMs handle Microservices reasonably well:** Most models showed a reasonably good understanding of microservices-based design, particularly the larger ones. These models often generated diagrams that needed refinement but provided a solid starting point.
- **RAG helps most of the time:** Using Retrieval-Augmented Generation (RAG) generally improved results. While a few models didn't benefit much or even declined in performance, most performed better when given a focused, concise RAG file that emphasized key microservice design principles.
- **Input format makes a difference:** When comparing full Software Requirements Specifications (SRS) with more concise Functional/Non-Functional Requirements (FR-NFR) lists, larger models handled full SRS documents better. Smaller models tended to perform worse with longer inputs but showed improved results with the shorter, more targeted FR-NFR format.
- **Objective metrics appear promising:** Although we didn't directly use objective metrics to evaluate the diagrams in this study, there was a noticeable correlation between these metrics and the subjective human evaluations. This points to the potential of automating evaluations using architecture-specific metrics in future work.
- **LLM-generated metrics vary in accuracy:** When models attempted to calculate the metrics on their own, the results were inconsistent. Hallucination was common, highlighting the need for careful interpretation and external validation.

4.4 MyCharts 2-Prompt Experiment

Lastly, we ran a final experiment, the two-prompt experiment described in Chapter 3.4. This small-scale test explored whether giving the model a follow-up prompt, based on the evaluation and metrics of its initial response, could help it produce a better second version. The idea came from reviewing results for *MyCharts*, where many initial outputs were close to correct and just needed minor adjustments. We wanted to see if a structured second prompt could guide the model to improve those responses more effectively.

4.4.1 Typical Cases

In this section, we showcase pairs of UML class diagrams generated by the LLMs, one from the initial prompt and the other from a follow-up, evaluation-informed prompt. These examples illustrate how the second prompt, guided by feedback and metrics from the first response, can enhance the structure, completeness and overall quality of the diagrams. By comparing these pairs, we aim to demonstrate the practical value of iterative prompting in refining architectural outputs.

Microservices | mistral | SRS | NoRAG | Response 1
The diagram in Figure 4.42 represents an adequate initial response, capturing most of the core features of the application. It demonstrates a reasonable distribution of functional responsibilities across multiple microservices, indicating a general grasp of the microservices architectural style. However, it falls short in two key areas: modeling interservice communication and incorporating dedicated databases for each microservice, both of which are essential aspects of a robust microservices design.
 *Figure 4.42: MyCharts 2-prompt Experiment - Mistral (Response 1)*
Microservices | mistral | SRS | NoRAG | Response 2
The diagram in Figure 4.43 shows a notable improvement, introducing an API Gateway to handle request routing while also clearly enhancing the modeling of interservice communication. This addition makes service interaction more explicit and structured.

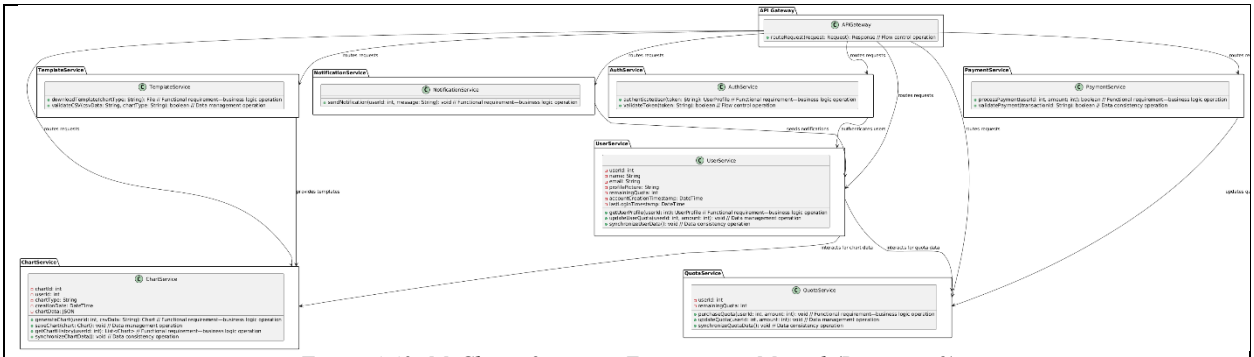


Figure 4.43: MyCharts 2-prompt Experiment - Mistral (Response 2)

Microservices | deepseek-r1 | SRS | NoRAG | Response 1
The diagram in Figure 4.44 shows a solid first attempt, capturing most of the key features of the application. It does a decent job of spreading responsibilities across different microservices, showing that the model understands the basics of microservices architecture. That said, it misses a couple of important points like inter-service communication and the use of separate databases for each microservice, which are both important for a strong microservices design.

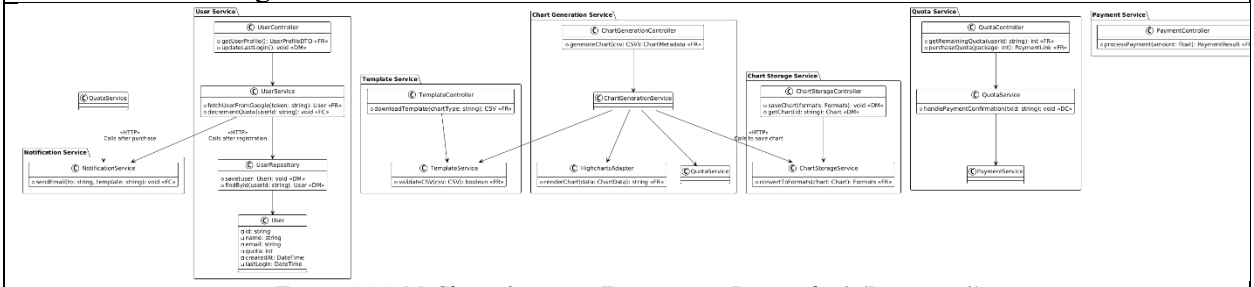


Figure 4.44: MyCharts 2-prompt Experiment - Deepseek-r1 (Response 1)

Microservices | deepseek-r1 | SRS | NoRAG | Response 2
The diagram in Figure 4.45 demonstrates a significant improvement in modeling inter-service communication. The addition of a publish-subscribe (pub-sub) messaging queue introduces asynchronous communication between services, enabling better scalability and decoupling. Alongside this, the inclusion of an API Gateway centralizes request routing and access control, reflecting a more realistic microservices architecture. These enhancements significantly elevate the architectural completeness and structure of the diagram.

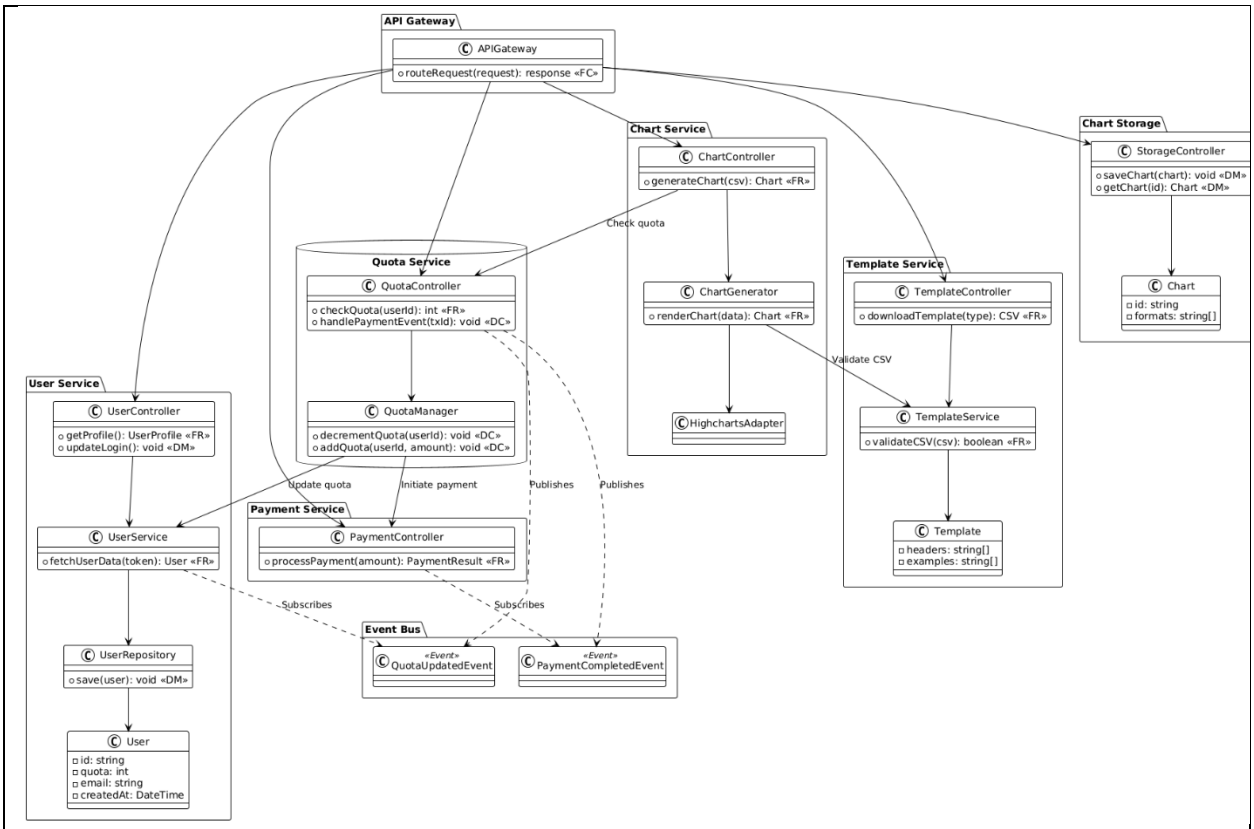


Figure 4.45: MyCharts 2-prompt Experiment - Deepseek-r1 (Response 2)

4.4.2 Evaluation Results

In this section, we present a series of charts illustrating how each model's output changed in response to a second, follow-up prompt. As shown in Figure 4.46, Figure 4.47, Figure 4.48, Figure 4.49, Figure 4.50 most models, like Claude Sonnet 3.7, Mistral-Online, DeepSeek-R1 and GPT-4o, showed noticeable improvement in their second responses, achieving higher scores across the evaluation criteria. However, model o1 experienced a slight decline in performance.

While this small-scale experiment isn't sufficient to draw broad conclusions, it does open the door for further research. It highlights the potential of using a standardized second prompt to refine LLM outputs. Currently, this process still requires manual evaluation of the first response and metric calculation, but it suggests a path toward a semi-automated refinement approach that could enhance architectural output quality.

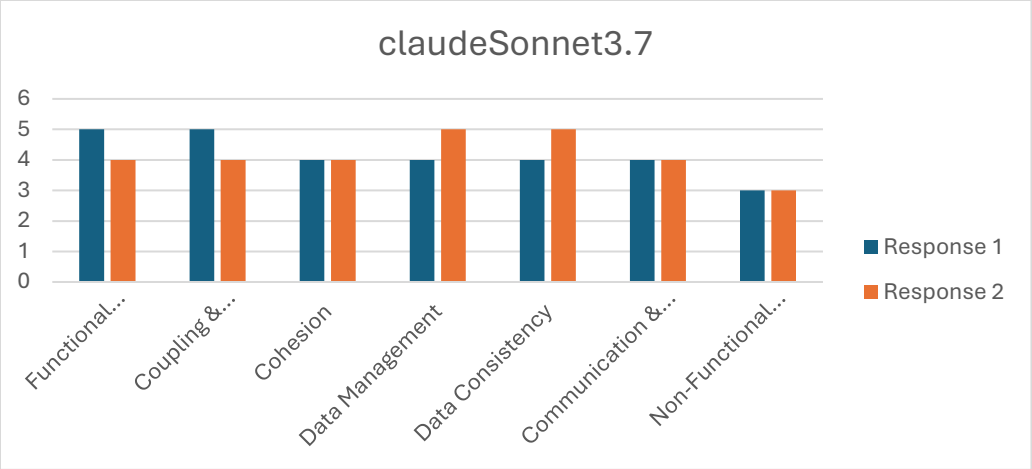


Figure 4.46: MyCharts 2-prompt Experiment (claudeSonnet3.7)

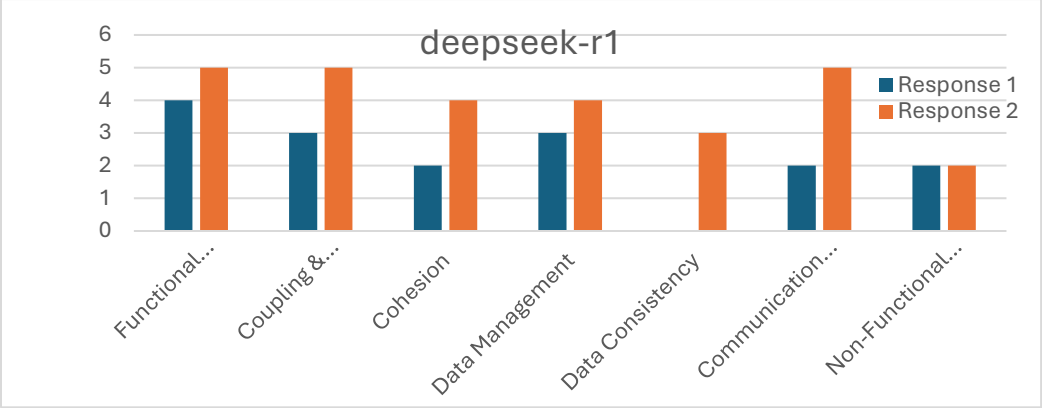


Figure 4.47: MyCharts 2-prompt Experiment (deepseek-r1)

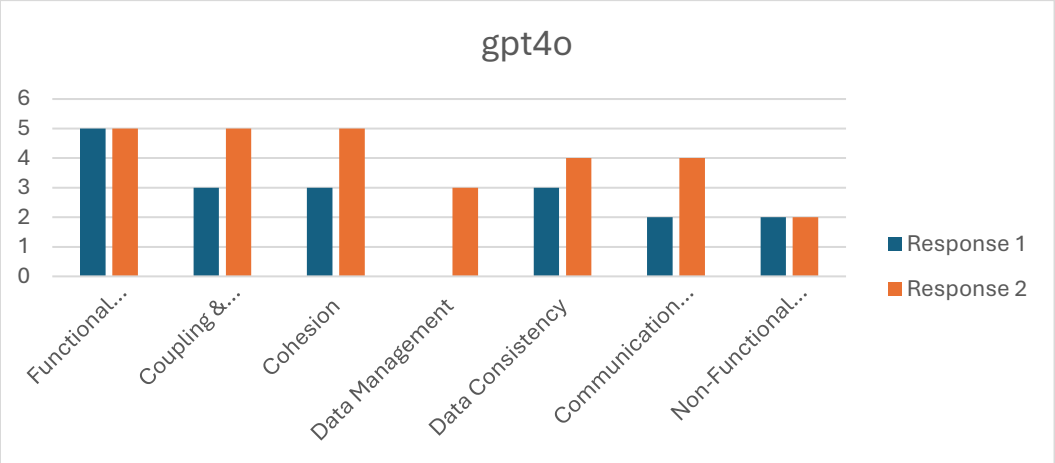


Figure 4.48: MyCharts 2-prompt Experiment (gpt4o)

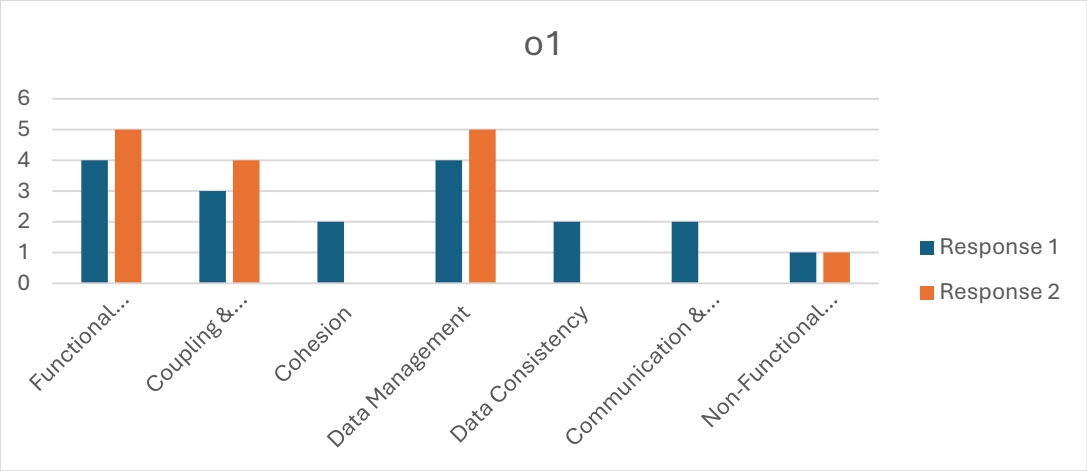


Figure 4.49: MyCharts 2-prompt Experiment (o1)

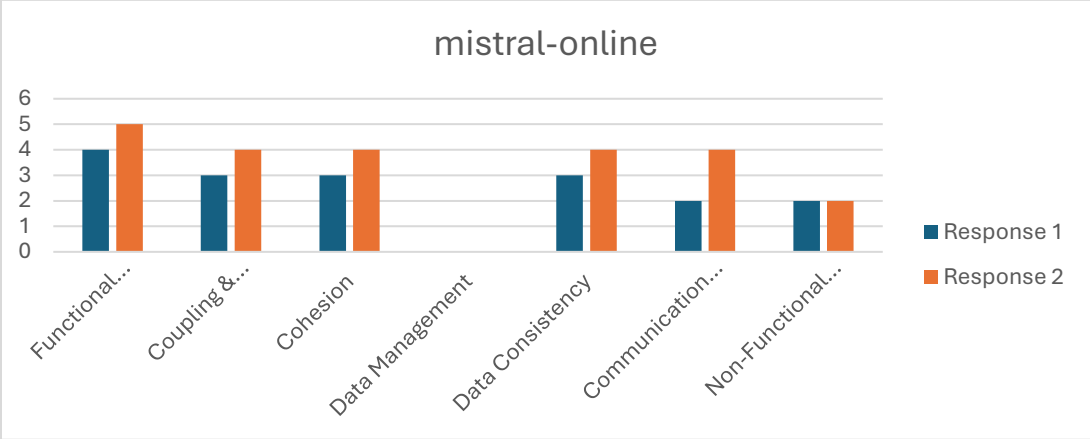


Figure 4.50: MyCharts 2-prompt Experiment (mistral-online)

5. Discussion

This thesis explored the capabilities and limitations of large language models (LLMs) in generating UML class/component diagrams that implement specific architectural patterns, including Client-Server, Three-Tier, MVC and Microservices. Through three structured experiments, we examined how different input formats such as functional and non-functional requirement lists, Software Requirements Specification (SRS) documents and the use of Retrieval-Augmented Generation (RAG) influence the quality of generated architectures. We then introduced a more complex software application and assessed LLM performance in designing more demanding architectures like Microservices and proposed new evaluation criteria and objective metrics specific to this pattern. Lastly, we tested a two-prompt approach, where a standardized follow-up prompt based on feedback from the initial output led to improved second responses, suggesting the potential for a semi-automated refinement process.

5.1 Conclusions

The results of this study highlight the promising potential of integrating LLMs into the design phase of the software development lifecycle. By examining input formats, output quality, architectural complexity, evaluation methods, model performance and refinement techniques, this research provides practical insights into how AI can support architectural design. These findings can help identify where LLMs add real value, recognize current limitations and develop strategies to effectively incorporate AI into real-world design workflows.

Reflecting on the research questions outlined in Chapter 2.5, we now attempt to address them based on the findings from this study.

1. Our evaluation of output formats revealed that when asking LLMs to create architectures as PlantUML documents, they consistently produce the higher quality architectural representations compared to other "diagram-as-code" formats. PlantUML documents, offer a balance of structure and readability that seems well-suited for LLMs. This insight provides a practical recommendation for teams integrating AI into their architecture workflows.

2. In terms of input representation, we found that structured Software Requirements Specification (SRS) documents can enhance LLM performance compared to plain-text requirements documents; notably, diagrams in SRS are also expressed as PlantUML text; this stands particularly for larger models, capable of benefiting from the extended context and rich structured information in SRS documents. This suggests that the quality and format of input data play a critical role in the output accuracy of AI-generated architectures. Simpler FR/NFR lists

may still be useful for smaller models, but they often lack the depth needed for more complex designs.

3. When tackling complex design challenges, specifically Microservices architectures, LLMs exhibited mixed performance. Larger, more capable models were generally more successful in decomposing complex requirements and applying core microservices principles, while smaller models often struggled. This emphasizes the need for LLM selection to be proportional to the architectural complexity of the problem domain.

4. Another part of this study is the introduction of specialized evaluation criteria and objective metrics specific to Microservices. This is a first attempt to fill gaps in existing assessment methods, and also to correlate subjective human evaluations with quantifiable architectural metrics, paving the way for more standardized benchmarking of LLM performance in future work.

5. We also observed that Retrieval-Augmented Generation (RAG) enhances design quality, particularly in larger models that can handle the increased context without being misled or confused. The RAG material seems to have a large impact for enhancing and not misleading the LLMs; our results indicate that concise and precise RAG materials with practical guidelines seem to work best.

6. Despite these advancements, the study also revealed some limitations. LLMs frequently hallucinate when asked to calculate the objective metrics in the diagrams themselves have created, often offering incorrect values. This underlines the importance of external objective validation and the risks of relying solely on AI-generated evaluations.

7. Finally, our experiment with iterative prompting showed encouraging results, suggesting that a structured follow-up prompt, which is informed by evaluation feedback, can refine and improve the quality of the architectural output. Although tested on a limited scale, this finding opens up possibilities for semi-automated refinement workflows that blend human oversight with AI-generated design.

Together, these findings provide a roadmap for both using LLMs in software architecture and understanding the conditions under which they are most effective. They also highlight areas for future exploration, including more robust automation pipelines, deeper integration of evaluation metrics and expanded use of conversational refinement techniques.

5.2 Future Work

While this study provides promising insights into the use of LLMs for software architecture generation, there remain several areas for future exploration. One important direction is scaling up the experiments, both in terms of the number of use cases and the diversity of architectural patterns examined. This would allow for broader generalizations and could

reveal whether the trends observed here hold across domains, project sizes and industry-specific applications.

Another promising area is the automation of the evaluation and refinement process. Currently, the two-prompt method introduced in this research relies on manually calculated metrics and human evaluations to guide refinement. A logical next step would be to integrate these processes into a semi-automated or fully automated pipeline that uses standardized, validated metrics to assess initial outputs and generate effective follow-up prompts. This could significantly streamline the use of LLMs in architectural design and make them more practical for integration into real-world development workflows.

Additionally, there is substantial room to improve hallucination mitigation, particularly in cases where LLMs are asked to calculate or reason about specific architectural metrics. Future studies could investigate techniques such as external knowledge integration or model fine-tuning to reduce inaccuracies. As LLMs continue to evolve, understanding how to reliably anchor their outputs in factual and context-aware reasoning will be critical to their successful application in software engineering.

6. Appendix

6.1 Appendix A (SRS_v1 for DCC Application)

Software Requirements Specification (SRS)

Dummy Coordinate Converter Application

1. Introduction

1.1 Introduction: Purpose of the Software

The Dummy Coordination Conversion (DCC) Application is an application that manages coordinate groups in Cartesian and Polar formats. The app allows users to input, convert, store, retrieve, modify, and delete coordinate groups.

1.2 Interfaces

1.2.1 User Interfaces

The user interacts with the application via a graphical user interface (GUI) developed using Java and the Swing framework. This interface includes all the necessary buttons and controls to implement the use cases outlined below.

2. References - Glossary

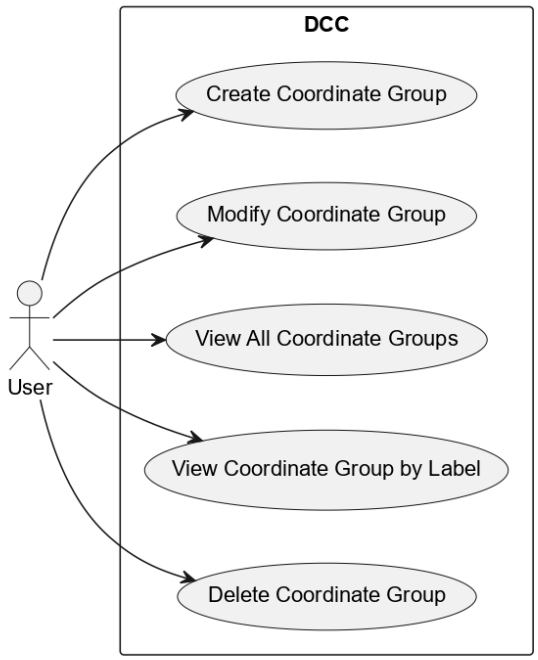
- **Cartesian Coordinates:**
  A type of coordinate representation that specifies a point's location using two values: x (horizontal position) and y (vertical position). Example: (x, y).
- **Polar Coordinates:**
  A type of coordinate representation that specifies a point's location using a radius (r) and an angle (θ), measured from the origin. Example: (r, θ).
- **Coordinate Group:**
  A collection of coordinates that includes both Cartesian and Polar representations.

3. Software Requirements Specifications

3.1 Use Cases

The user should be able to perform the following actions: create a coordinate group, modify a coordinate group, delete a coordinate group, view all coordinate groups, and view a specific coordinate group selected by its label. These use cases are illustrated in the use case diagram provided below. Additionally, each use case is elaborated in detail in the subsequent sections.

Use Case Diagram



3.1.1 Use Case 1: (Create Coordinate Group)

3.1.1.1 Preconditions for Execution

The sole prerequisite for executing this use case is that the application must be running.

3.1.1.2 Execution Environment

The execution environment for this use case is the graphical user interface (GUI). This interface provides all the required buttons and controls to facilitate user interactions.

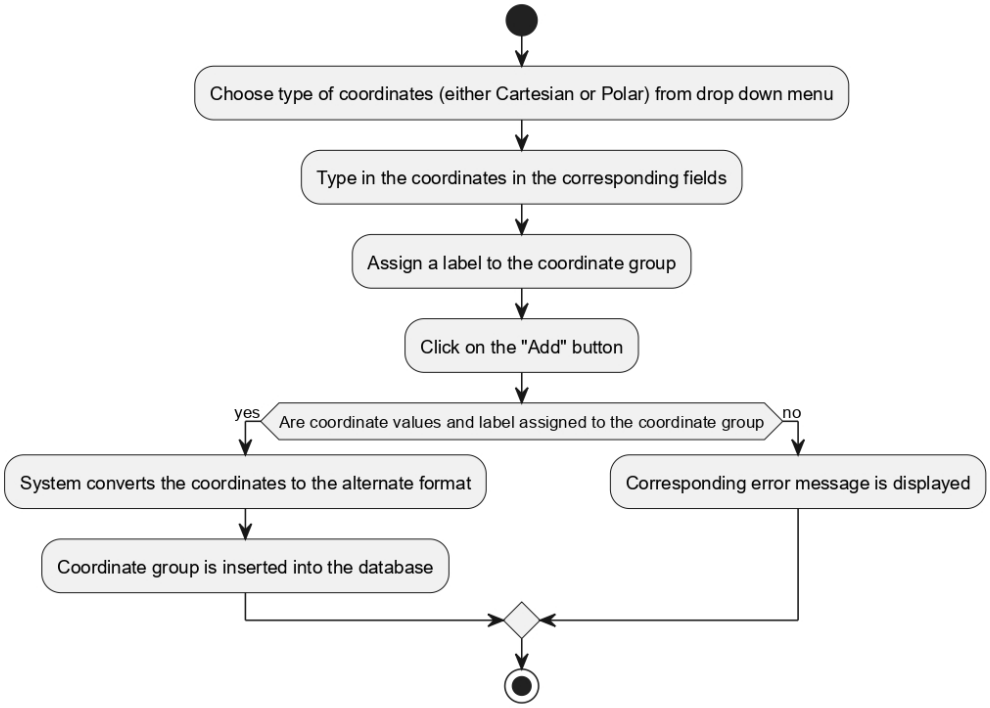
3.1.1.3 Input Data

The input data consists of the source coordinate type (either Cartesian or Polar) and the corresponding values: x and y for Cartesian coordinates, or r and θ for Polar coordinates. Also, a unique label to identify the coordinate group.

3.1.1.4 Sequence of Actions - Desired Behavior

The flow of the program in this use case is described by the following actions:
Step 1: Choose source coordinate type from drop down menu (either Cartesian or Polar).
Step 2: Fill in the values of the coordinates in the corresponding fields.
Step 3: Assign a label to the coordinate group by filling the corresponding field.
Step 4: Press the "Add" button.

Activity Diagram (Create Coordinate Group)



3.1.1.5 Output Data

The newly created coordinate group is stored in the system's database; this includes the source coordinates and the coordinates converted to the alternate format, as well as the label.

3.1.2 Use Case 2: (Modify Coordinate Group)

3.1.2.1 Preconditions for Execution

a) an active connection to the database
b) the existence of the coordinate group that is intended to be modified.

3.1.2.2 Execution Environment

The execution environment for this use case is the graphical user interface (GUI). This interface provides all the required buttons and controls to facilitate user interactions.

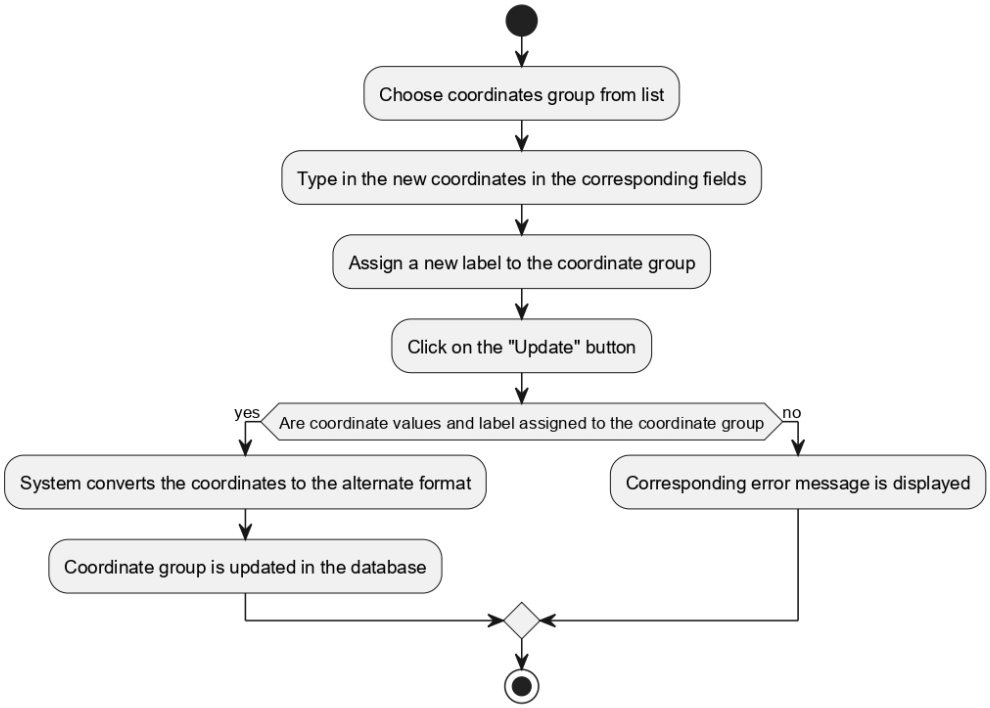
3.1.2.3 Input Data

The input data consists of a selected coordinate group, the new label, the source coordinate type (either Cartesian or Polar) and the new corresponding values: x and y for Cartesian coordinates, or r and θ for Polar coordinates.

3.1.2.4 Sequence of Actions - Desired Behavior

The flow of the program in this use case is described by the following actions:
Step 1: Choose coordinate group from a list.
Step 2: Fill in the new values of the coordinates in the corresponding fields.
Step 3: Assign the new label in the corresponding field.
Step 4: Press the "Update" button in the User Interface.

Activity Diagram (Modify Coordinate Group)



3.1.2.5 Output Data

The modified coordinate group is stored in the system's database; this includes the source coordinates and the coordinates converted to the alternate format, as well as the label.

3.1.3 Use Case 3: (View All Coordinate Groups)

3.1.3.1 Preconditions for Execution

a) an active connection to the database

3.1.3.2 Execution Environment

The execution environment for this use case is the graphical user interface (GUI). This interface provides all the required buttons and controls to facilitate user interactions.

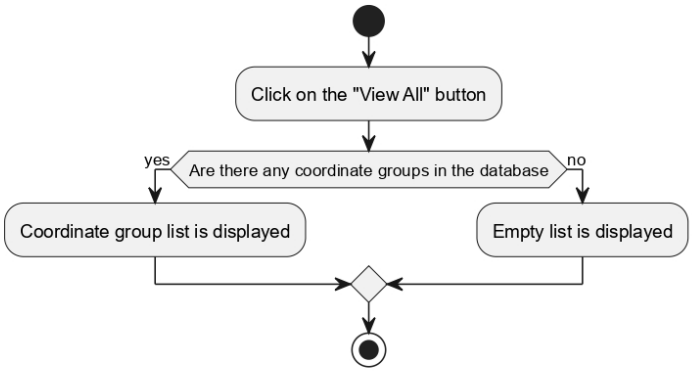
3.1.3.3 Input Data

The is no input data.

3.1.3.4 Sequence of Actions - Desired Behavior

The flow of the program in this use case is described by the following actions:
Step 1: Press the "View All" button in the User Interface.
Step 2: Retrieve all stored coordinate groups from the database.
Step 3: View the list of coordinates.

Activity Diagram (View All Coordinate Groups)



3.1.3.5 Output Data

The output data is the list of the coordinate groups in the database.

3.1.4 Use Case 4: (View Coordinate Groups by Label)

3.1.4.1 Preconditions for Execution

a) an active connection to the database

3.1.4.2 Execution Environment

The execution environment for this use case is the graphical user interface (GUI). This interface provides all the required buttons and controls to facilitate user interactions.

3.1.4.3 Input Data

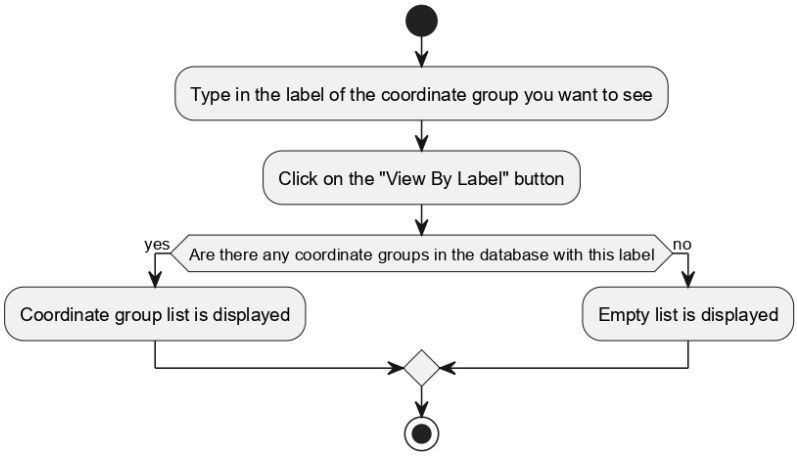
The input data consists of label to be used as filter for selecting coordinate groups from the database.

3.1.4.4 Sequence of Actions - Desired Behavior

The flow of the program in this use case is described by the following actions
Step 1: Type in the label in the corresponding field in the User Interface.

Step 2: Press the "Find by Label" button.
Step 2: Retrieve the coordinate groups from the database that satisfy the search criteria.
Step 3: View the retrieved groups of coordinates in a list.

Activity Diagram (View Coordinate Groups by Label)



3.1.4.5 Output Data

The output data is the list of the coordinate groups in the database that satisfy the selection filter.

3.1.5 Use Case 5: (Delete Coordinate Groups)

3.1.5.1 Preconditions for Execution

a) an active connection to the database
b) there is at least one coordinates group in the database

3.1.5.2 Execution Environment

The execution environment for this use case is the graphical user interface (GUI). This interface provides all the required buttons and controls to facilitate user interactions.

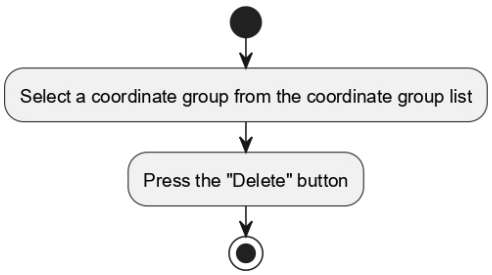
3.1.5.3 Input Data

The input data consists of the coordinate group that is selected from the list.

3.1.5.4 Sequence of Actions - Desired Behavior

The flow of the program in this use case is described by the following actions
Step 1: Select a coordinate group from the coordinate group list.
Step 2: Press the "Delete" button.

Activity Diagram (Delete Coordinate Group)



3.1.5.5 Output Data

The selected coordinate group is deleted from the system's database.

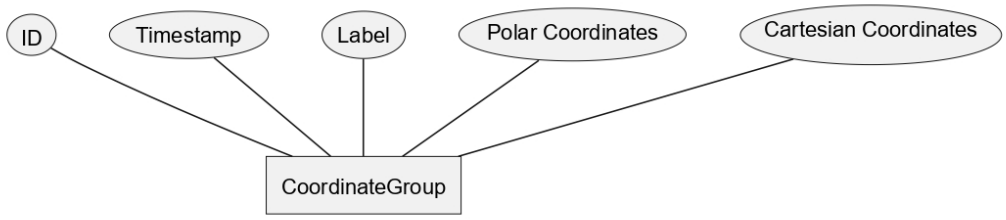
3.2 Data Requirements

3.2.1 Data Organization Organization Requirements

The database must be designed to store the following attributes for each coordinate group:

- Auto-generated unique primary key (integer)
- Timestamp (automatically assigned the date and time of group creation)
- Unique user-defined label for identification
- Coordinates in Polar format (r, θ)
- Coordinates in Cartesian format (x,y)

The following conceptual Entity Relationship Diagram describes the high-level design of the database.



3.2.2 Data Access Requirements

The system must be designed to store all data in a MySQL database. The application will connect to the database using a TCP/IP connection.

6.2 Appendix B (SRS_v2 for DCC Application)

Software Requirements Specification (SRS)

Dummy Coordinate Converter Application

1. Introduction

1.1 Purpose

The Dummy Coordination Conversion Application (DCC app) is an application that manages coordinate groups in Cartesian and Polar formats. The app allows users to input, convert, store, retrieve, modify, and delete coordinate groups.

1.2 Scope

This application serves as a lightweight tool for managing coordinate groups, used in several calculations by students, engineers, educators, etc.. It provides a GUI developed with Java Swing and supports essential operations such as coordinate conversion, storage, and retrieval. This document contains the requirements specification of the DCC app.

1.3 Definitions, Acronyms, and Abbreviations

- **Cartesian Coordinates:**
  A type of coordinate representation that specifies a point's location using two values: x (horizontal position) and y (vertical position). Example: (x, y).
- **Polar Coordinates:**
  A type of coordinate representation that specifies a point's location using a radius (r) and an angle (θ), measured from the origin. Example: (r, θ).
- **Coordinate Group:**
  A collection of coordinates that includes both Cartesian and Polar representations.

- **GUI**: Graphical User Interface.

1.4 References

- Polar coordinate system: https://en.wikipedia.org/wiki/Polar_coordinate_system
- Cartesian coordinate system: https://en.wikipedia.org/wiki/Cartesian_coordinate_system
- Java runtime environment: https://en.wikipedia.org/wiki/Java_(software_platform)#Java_Runtime_Environment
- Java Swing Framework: https://docs.oracle.com/javase/8/docs/technotes/guides/swing/
- MySQL Documentation: https://dev.mysql.com/doc/

1.5 Overview

This document outlines the software requirements for the DCC Application, detailing functional and non-functional requirements, data organization, and interface expectations. It is structured to facilitate both development and validation processes.

2. System Overview

2.1 Product Perspective

The DCC Application operates as a standalone desktop application, that runs in all modern operating systems (windows, macos and linux). It uses a MySQL database for backend storage and Java Swing for the user interface. Its primary goal is to simplify coordinate system transformations and management.

2.2 Product Functions

- Create, view, modify, and delete coordinate groups.
- Convert coordinates between Cartesian and Polar systems.
- Store coordinate groups persistently in a database.

2.3 User Characteristics

The intended users are students, engineers, educators, that need to convert between coordinate systems. No special technical skills are required to operate the software.

2.4 Operating Environment

- **Hardware**: Any computer that can run a JRE.
- **Software**: Java Runtime Environment (JRE) 8 or higher, MySQL Server 8.0 or higher.
- **Network**: Required for database access.

2.5 Assumptions and Dependencies

- The application requires a connection to a MySQL database.
- The Java Swing library is required for running the application's GUI
- A JDBC connector is required for database communication.

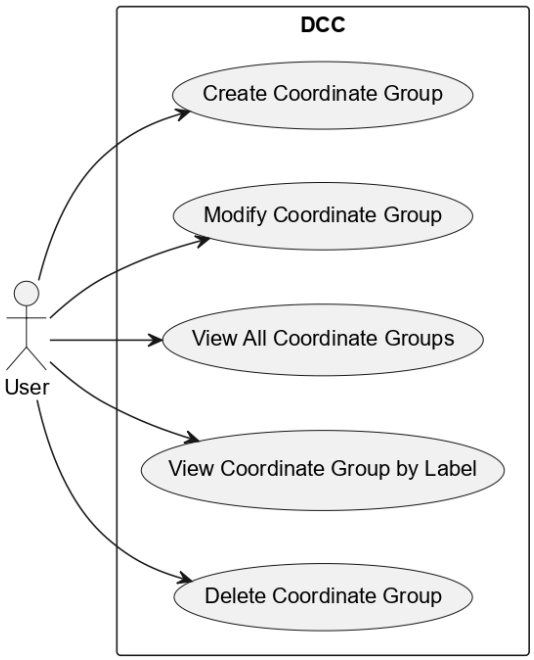
3. Functional Requirements

3.1 Functional Requirements Specification

1. Define cartesian coordinates as a pair (x, y) of high-precision numbers
2. Define polar coordinates as a pair (r, θ) of high-precision numbers
3. Convert cartesian (x, y) to polar (r, θ) coordinates
4. Convert polar (r, θ) to cartesian (x, y) coordinates
5. Create a unique auto-generated integer identifier (id) for the coordinate group edited, add a timestamp, and save the label and the values of the coordinates group as a new record into the database
6. Create a list of all saved coordinate group records. For each record, show: id, label, cartesian coordinate values, polar coordinate values, datetime added or modified), sorted by id
7. Update a coordinate group record with new values entered by the user and save the updated record of the coordinates group into the database
8. Delete a coordinate group record from the database
9. Provide error handling for invalid inputs.

3.2 Use Cases

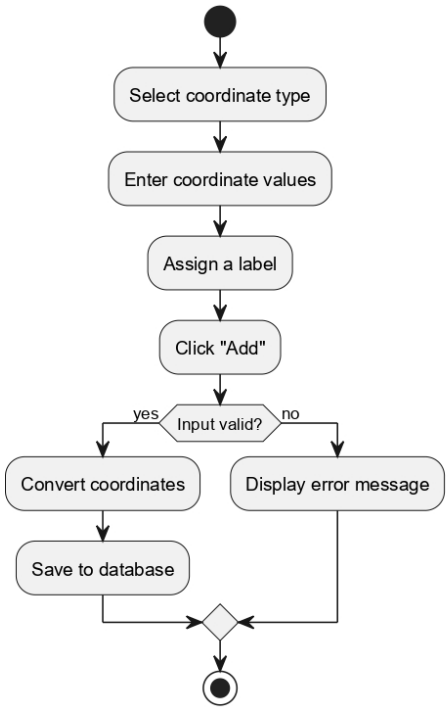
3.2.1 Use Case Diagram



3.2.2 Use Case 1: Create Coordinate Group

- **Use Case ID**: UC1
- **Actors**: User
- **Execution environment:** The execution environment for this use case is the graphical user interface (GUI). This interface provides all the required buttons and controls to facilitate user interactions.
- **Preconditions**: The application must be running.
- **Input Data**: The input data consists of the source coordinate type (either Cartesian or Polar) and the corresponding values: x and y for Cartesian coordinates, or r and θ for Polar coordinates. Also, a unique label to identify the coordinate group.
- **Main Flow**:
  Step 1: Choose source coordinate type from drop down menu (either Cartesian or Polar).
  Step 2: Fill in the values of the coordinates in the corresponding fields.
  Step 3: Assign a label to the coordinate group by filling the corresponding field.
  Step 4: Press the "Add" button.
  Step 5: The app converts coordinates to the alternate format and stores them in the database.

- **Alternate Flow**:
  - If the input is invalid, the system displays an error message.
- **Postconditions**:
  a) The new coordinate group is saved in the database.
- **Related Diagrams**: UML Activity Diagram (see below).



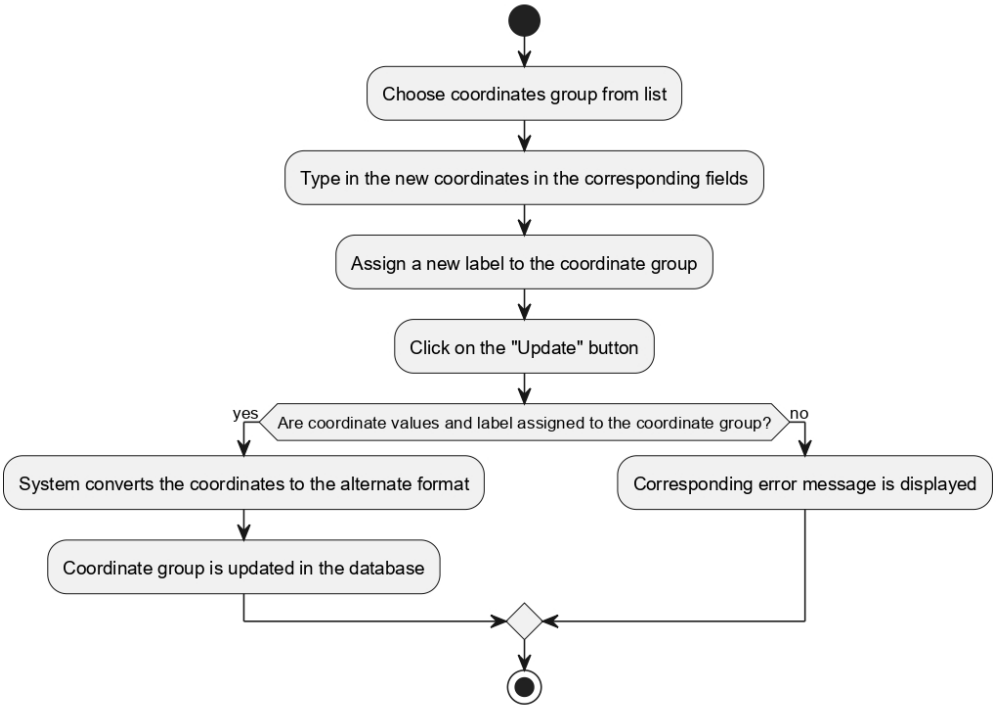
- **Output Data**: The newly created coordinate group is stored in the system's database; this includes the source coordinates and the coordinates converted to the alternate format, as well as the label.

3.2.3 Use Case 2: Modify Coordinate Group

- **Use Case ID**: UC2
- **Actors**: User
- **Execution environment:** The execution environment for this use case is the graphical user interface (GUI). This interface provides all the required buttons and controls to facilitate user interactions.
- **Preconditions**:
  a) an active connection to the database

b) the existence of the coordinate group that is intended to be modified.

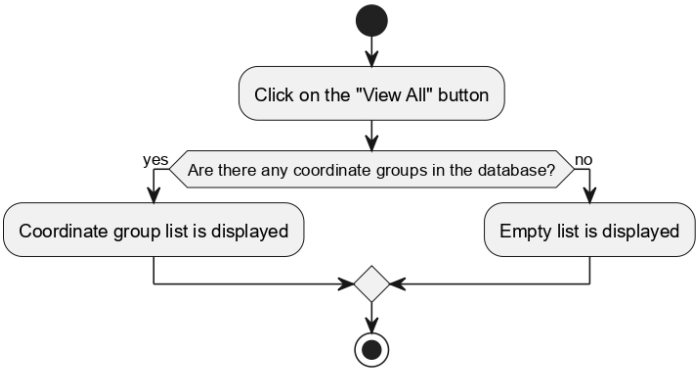
- **Input Data**: The input data consists of a selected coordinate group, the new label, the source coordinate type (either Cartesian or Polar) and the new corresponding values: x and y for Cartesian coordinates, or r and θ for Polar coordinates.
- **Main Flow**:
  Step 1: Choose coordinate group from a list.
  Step 2: Fill in the new values of the coordinates in the corresponding fields.
  Step 3: Assign the new label in the corresponding field.
  Step 4: Press the "Update" button in the User Interface.
- **Alternate Flow**:
  - If input is invalid, the system displays an error message.
- **Postconditions**:
  a) The coordinate group is updated in the database.
- **Related Diagrams**: UML Activity Diagram (see below).



- **Output Data**: The modified coordinate group is stored in the system's database; this includes the source coordinates and the coordinates converted to the alternate format, as well as the label.

3.2.4 Use Case 3: View All Coordinate Groups

- **Use Case ID**: UC3
- **Actors**: User
- **Execution environment:** The execution environment for this use case is the graphical user interface (GUI). This interface provides all the required buttons and controls to facilitate user interactions.
- **Preconditions**:
  a) an active connection to the database
- **Input Data**: The is no input data.
- **Main Flow**:
  Step 1: Press the "View All" button in the User Interface.
  Step 2: Retrieve all stored coordinate groups from the database.
  Step 3: View the list of coordinates.
- **Postconditions**:
  a) A list of coordinate groups is displayed.
- **Related Diagrams**: UML Activity Diagram (see below).

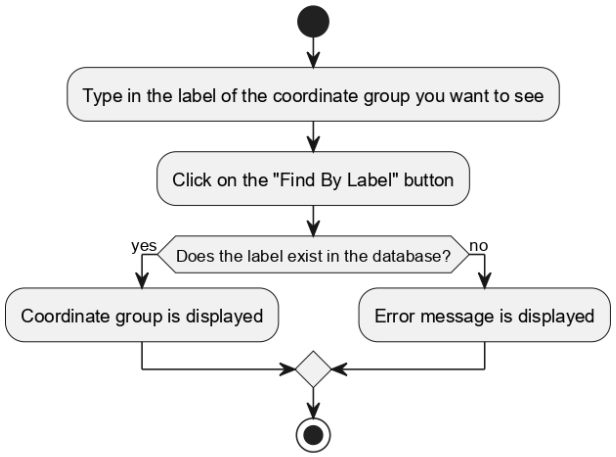


- **Output Data**: The output data is the list of the coordinate groups in the database.

3.2.5 Use Case 4: View Coordinate Group by Label

- **Use Case ID**: UC4
- **Actors**: User
- **Execution environment:** The execution environment for this use case is the graphical user interface (GUI). This interface provides all the required buttons and controls to facilitate user interactions.

- **Preconditions**:
  a) an active connection to the database
- **Input Data**: The input data consists of label to be used as filter for selecting coordinate groups from the database.
- **Main Flow**:
  Step 1: Type in the label in the corresponding field in the User Interface.
  Step 2: Press the "Find by Label" button.
  Step 2: Retrieve the coordinate groups from the database that satisfy the search criteria.
  Step 3: View the retrieved groups of coordinates in a list.
- **Postconditions**:
  a) The selected coordinate group is displayed.
- **Related Diagrams**: UML Activity Diagram (see below).

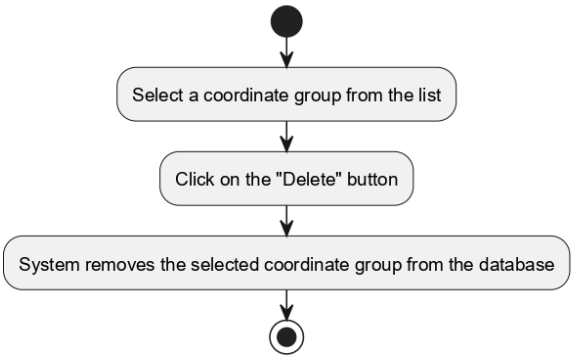


- **Output Data**: The output data is the list of the coordinate groups in the database that satisfy the selection filter.

3.2.6 Use Case 5: Delete Coordinate Group

- **Use Case ID**: UC5
- **Actors**: User
- **Execution environment:** The execution environment for this use case is the graphical user interface (GUI). This interface provides all the required buttons and controls to facilitate user interactions.
- **Preconditions**:
  a) an active connection to the database
  b) there is at least one coordinates group in the database

- **Input Data**: The input data consists of the coordinate group that is selected from the list.
- **Main Flow**:
  Step 1: Select a coordinate group from the coordinate group list.
  Step 2: Press the "Delete" button.
  Step 3: System removes the selected coordinate group from the database.
- **Postconditions**:
  a) The selected coordinate group is deleted.
- **Related Diagrams**: UML Activity Diagram (see below).



- **Output Data**: The selected coordinate group is deleted from the system's database.

4. Non-Functional Requirements

4.1 Performance Requirements

- The system shall process user inputs and database operations within 2 seconds.

4.2 Usability Requirements

- The GUI shall be simple and comprehensive, with clear labels and buttons.

4.3 Security Requirements

- The system shall authenticate to the database using a secure username and password.

4.4 Availability Requirements

- The app should be able to maintain 90% uptime during operation hours.

4.5 Scalability Requirements

- The system shall support up to 100,000 (one hundrend thousand) coordinate groups without performance degradation.

4.6 Compliance Requirements

- The system shall adhere to applicable data handling and privacy laws.

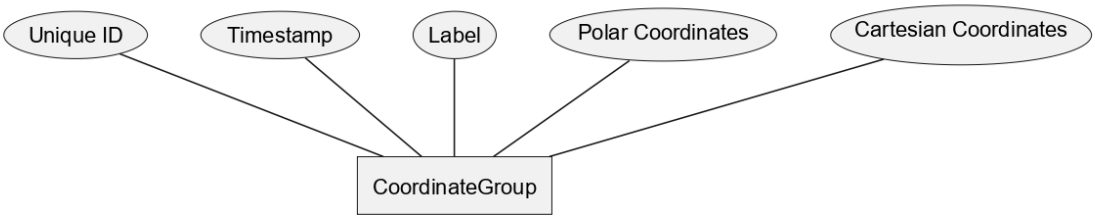
5. Data Requirements

5.1 Data Models

Each coordinate group shall include:

- Auto-generated unique primary key (integer)
- Timestamp (automatically assigned the date and time of group creation)
- Unique user-defined label for identification
- Coordinates in Polar format (r, θ)
- Coordinates in Cartesian format (x,y)

The following conceptual Entity Relationship Diagram describes the high-level design of the database.



5.2 Data Storage

- Data shall be stored in a MySQL database with proper indexing for performance.

5.3 Data Access

- The system shall use a JDBC connection for database access.

5.4 Data Validation

- Coordinate values must adhere to valid ranges (e.g., θ in [0, 360] degrees).

6. Interface Requirements

6.1 User Interfaces

- The GUI shall include input fields, dropdowns, and buttons for all core functionalities.
- The UI shall include:
  - Main menu with navigation options.
  - Forms for creating and modifying coordinate groups.
  - Display list for viewing coordinate groups.

6.2 External Interfaces

- The system shall interface with a MySQL database via JDBC.

7. System Constraints

7.1 Design Constraints

- The system shall use Java Swing for the GUI.
- The database must be MySQL.

7.2 Environmental Constraints

- The application must run on systems with Java 8 or later JRE installed.

8. Verification and Validation

8.1 Verification Plan

- Unit tests shall verify all functional requirements.
- Integration tests shall validate the integration with the DBMS service.

8.2 Validation Plan

- User acceptance testing shall confirm that the system meets the users' expectations for simplicity and efficiency in the user interface.

8.3 Traceability Matrix

- Each requirement shall be mapped to test cases to ensure coverage.

9. Appendices

9.1 Glossary

- **GUI**: Graphical User Interface
- **JDBC**: Java Database Connectivity

9.2 References

- See Section 1.4 for detailed references.

9.3 Change History

- Version 1.0: Initial release of the DCC Application SRS.

6.3 Appendix C (SRS for MyCharts Application)

Software Requirements Specification (SRS)

MyCharts Application

1. Introduction

1.1 Purpose

The MyCharts Application is a web-based service designed to enable users with minimal technical expertise to generate, manage, and download charts in various formats. It simplifies chart creation by providing templates for source data, supporting data uploads. It uses the Highcharts library for chart generation.

1.2 Scope

This application allows users to:

- Download CSV templates for supported chart types.
- Upload CSV files to generate charts.
- Save and download charts in PDF, PNG, SVG, and HTML formats.
- Purchase quotas for chart creation.
- View and download generated charts.

1.3 Definitions, Acronyms, and Abbreviations

- **CSV**: Comma-Separated Values file format.
- **Quota**: A limit on the number of charts a user can create.
- **Highcharts**: JavaScript library used for chart rendering.

1.4 References

- Highcharts Library: https://www.highcharts.com
- Google Authentication API: https://developers.google.com/identity
- CSV Format Specification: https://tools.ietf.org/html/rfc4180

1.5 Overview

This document outlines functional and non-functional requirements, use cases, and interface specifications for the MyCharts Application.

2. System Overview

2.1 Product Perspective

MyCharts is a SaaS (Software as a Service) standalone web application. It will be offered online and produce revenue by selling credits for chart generation.

2.2 Product Functions

1. User authentication via Google accounts.
2. Download CSV templates for chart types.

3. Upload CSV files to generate charts.
4. Save charts in PDF, PNG, SVG, and HTML formats.
5. Display and manage user-generated charts.
6. Charge for quotas needed to create charts.

2.3 User Characteristics

Primary users include educators, students, and professionals needing simple chart generation. No coding or technical skills are required.

2.4 Operating Environment

- **Frontend**: Modern browsers (Chrome, Firefox, Safari).
- **Backend**: Application Server with Node.js or Python runtime and database server(s) as needed.
- **Network**: Internet connectivity required for service offering.

2.5 Assumptions and Dependencies

- Google OAuth is available for user authentication.
- Highcharts library is accessible for chart rendering.
- Users follow the CSV structure.

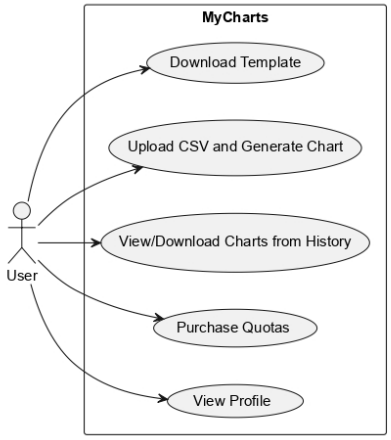
3. Functional Requirements

3.1 Functional Requirements Specification

1. Authenticate users via Google accounts.
2. Allow download of CSV templates for 3 supported chart types (Basic line, Line with annotations, Basic column).
3. Upload CSV file with the user data for chart generation.
4. Validate uploaded CSV files against the template structure, data types, and mandatory data existence.
5. Generate charts with data from the CSV file uploaded by the user, using Highcharts.
6. Save charts to the server in PDF, PNG, SVG, and HTML formats.
7. Display a dashboard with the history of user-generated charts, including chart previews.
8. Download selected chart type.
9. Charge quotas for chart creation.
10. Allow users to delete or download charts.
11. Sell quotas and receive payment by a payment gateway
12. Maintain user profiles containing the following data: Name (from Google account), profile picture (from Google account), email (from Google account), remaining quota, account creation timestamp, last login timestamp.
13. Display user's profile info, including remaining quotas.

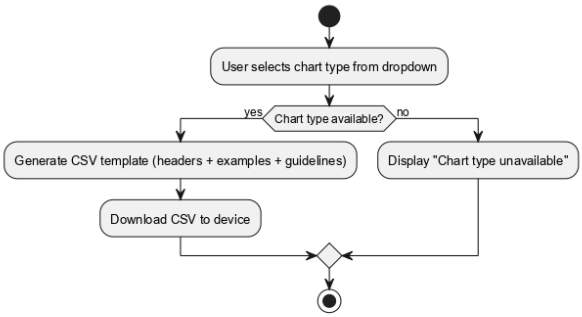
3.2 Use Cases

3.2.1 Use Case Diagram



3.2.2 Use Case 1: Download CSV Template

- **Actors**: User
- **Preconditions**:
  a) User is logged in.
- **Main Flow**:
  i. User navigates to the "Create Chart" section and selects a chart type from a dropdown menu.
  ii. System displays the 3 supported chart types.
  iii. User clicks "Download Template" for the selected chart type.
  iv. System provides a CSV file for each chart type, containing:
    - Predefined column headers matching the chart type.
    - Sample data rows, to be replaced by the user's actual data.
    - Chart formatting parameters.
  v. CSV file is downloaded to the user's device.
- **Post-conditions**:
  a) CSV template is saved locally on the end-user's machine for data entry.
- **Activity Diagram**:

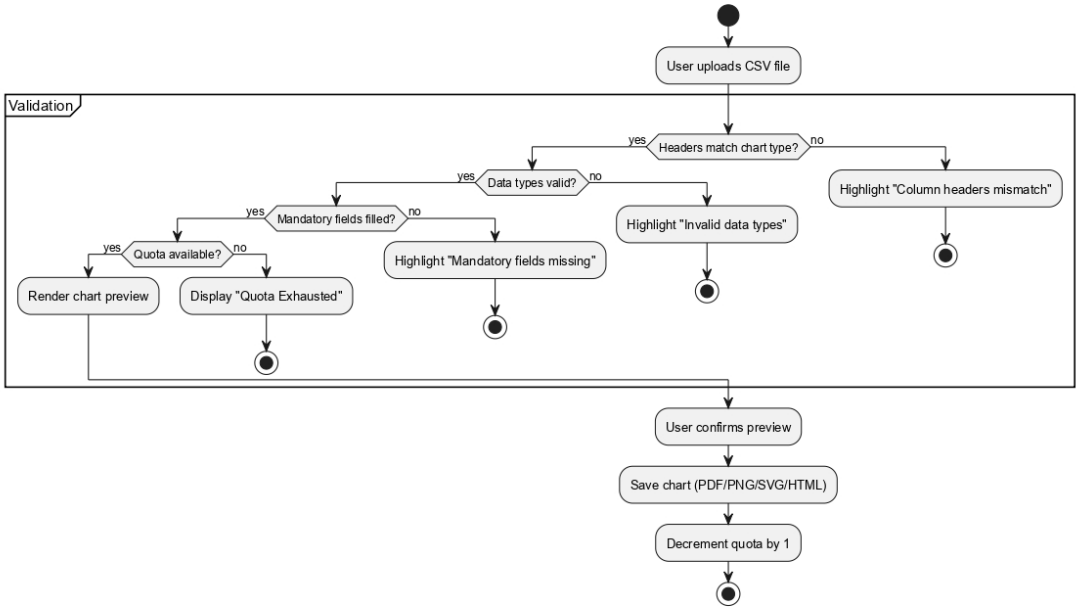


3.2.3 Use Case 2: Upload CSV and Generate Chart

- **Actors**: User
- **Preconditions**:
  a) User has a valid CSV file formatted per the template.

b) User has remaining quota for chart creation.

- **Main Flow**:
  i. User navigates to the "Generate Chart" section and uploads a CSV file.
  ii. System performs validation checks:
    - Column headers match the selected chart type.
    - Data types are correct (e.g., numeric values for data).
    - No empty mandatory fields.
  iii. System renders a preview of the chart using Highcharts.
  iv. User confirms the preview.
  v. System saves the chart to the server in all supported formats (PDF, PNG, SVG, HTML).
  vi. User's quota is decremented by 1.
- **Alternate Flows**:
  - **Invalid CSV**: System highlights errors (e.g., "Column 'Revenue' missing") and blocks submission.
  - **Quota Exhausted**: System displays "Quota Limit Reached: Upgrade to create more charts."
- **Post-conditions**:
  a) Chart is stored on the server.
  b) Chart appears in the user's dashboard.
- **Activity Diagram**:

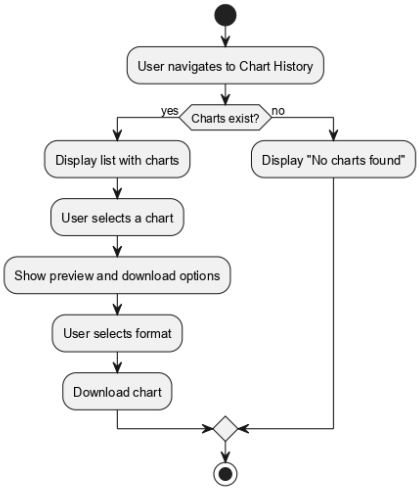


3.2.4 Use Case 3: View/Download Chart from History

- **Actors**: User
- **Preconditions**:
  a) User is logged in.
  b) At least one chart has been previously generated by the logged in user.
- **Main Flow**:
  i. User navigates to the "Chart History" section in the dashboard.
  ii. System displays a list of all previously generated charts with timestamps.
  iii. User selects a chart from the list.
  iv. System displays a preview and provides download options (PDF, PNG, SVG, HTML).
  v. User selects a format to download the chart.
- **Alternate Flow**:

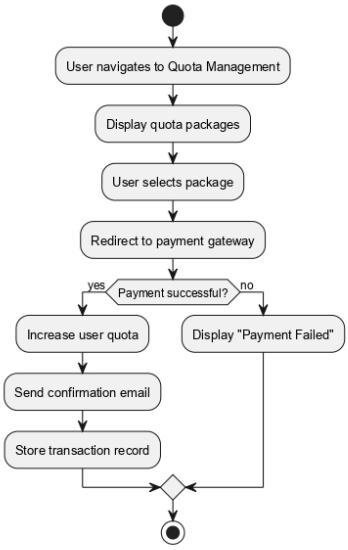
- If no charts exist, the system displays a message: "No charts found in history."

- **Post-conditions**:
  a) The selected chart is downloaded in the chosen format.
- **Activity Diagram**:



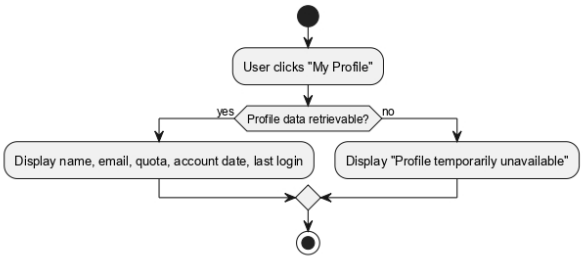
3.2.5 Use Case 4: Purchase Additional Quotas

- **Actors**: User
- **Preconditions**:
  a) User is logged in.
  b) Payment gateway integration is operational.
- **Main Flow**:
  i. User navigates to the "Quota Management" section in the dashboard.
  ii. User selects the number of quotas to purchase and confirms the purchase.
  iii. System redirects the user to a secure payment gateway (e.g., Stripe/PayPal).
  iv. User completes the payment process.
  v. System validates the payment and increases the user's quota limit accordingly.
  vi. System sends a confirmation email with a receipt.
- **Alternate Flows**:
  - **Payment Failure**: If payment fails, the system retains the original quota and notifies the user.
  - **Partial Payment**: System cancels the transaction and restores the initial quota state.
- **Post-conditions**:
  a) The user's quota limit is updated.
  b) A transaction record is stored in the system.
- **Activity Diagram**:



3.2.6 Use Case 5: View My Profile

- **Actors**: User
- **Preconditions**:
  a) User is logged in.
- **Main Flow**:
  i. User clicks the "My Profile" button in the dashboard navigation menu.
  ii. System retrieves and displays the following profile details:
    - Name and profile picture (from Google account).
    - Email address associated with the Google account.
    - Current chart creation quota (e.g., "15/20 charts remaining").
    - Account creation date.
    - Last login timestamp.
  iii. User reviews the displayed information.
- **Alternate Flows**:
  - **Profile Data Unavailable**: If the system cannot retrieve data (e.g., server error), it displays "Profile temporarily unavailable."
- **Post-conditions**:
  a) User views their profile details.
- **Activity Diagram**:



4. Non-Functional Requirements

4.1 Performance

- Chart generation completes within **3 seconds**.

4.2 Usability

- **Intuitive UI** with guided workflows (e.g., tooltips, validation hints) to assist non-technical users.

4.3 Security

- **SSL encryption**: Data transmitted ove the internet is encrypted using SSL.

4.4 Availability

- The service should be available 90% of the time on a daily basis.
- The system should prioritize **Availability over Consistency** during network partitioning.
- While in network partitioning, although temporary data inconsistencies may occur, as many as possible services should remain fully operational.

4.5 Scalability

- Support **1,000 concurrent users** with linear performance scaling via cloud-based auto-scaling infrastructure.

5. Data Requirements

5.1 Data Models

- **User Profile**: Store the following attributes for every user:
  - Name (from Google account)
  - Profile picture (from Google account)
  - Email (from Google account)
  - Remaining quota
  - Account creation timestamp
  - Last login timestamp
- **Chart Metadata**: Chart ID, type, creation date, Highcharts json chart object.

5.2 Data management

Data will be stored in databases (e.g., PostgreSQL, MySQL, MongoDB, etc), as required by the architecture to be implemented.

5.3 Data consistency

- After network partitioning, data should be synchronized to achieve a consistent state across the system's components.

6. Interface Requirements

6.1 User Interfaces

Web app, offering a frontend containing the following elements:

1. **Landing Page**

- Service description and features

- Sample chart gallery
- Pricing information
- Call-to-action buttons for login-registration

2. **Authentication Page**

- Google OAuth login button

3. **Profile Dashboard**

- User details from Google account
- Current quota status
- Account statistics (creation date, last login)
- Profile picture display

4. **Chart Management**

- View of previously generated charts
- Preview thumbnails
- Download/delete actions

5. **Chart Generator**

- Chart type selector
- CSV template downloader
- File upload zone with validation
- Real-time chart preview
- Format selection for export

6. **Quota Store**

- Quota selection
- Pricing display
- Payment gateway integration

6.2 External Interfaces

- Google OAuth for authentication.

7. System Constraints

7.1 Design Constraints

- Frontend: React Framework.
- Backend: Use of REST APIs, API gateways, Load balancer as needed.

7.2 Environmental Constraints

- Requires modern browser support.

8. Verification and Validation

8.1 Verification Plan

- Unit tests for chart generation logic.
- Integration tests for Google OAuth.

8.2 Validation Plan

- User testing for ease of CSV upload and chart customization.

9. Appendices

9.1 Glossary

- **Quota**: Credits remaining for chart generation per user.
- **CSV**: Data file format for chart input.

7. References

[1] Shaveta, "A review on machine learning," *International Journal of Science and Research Archive*, vol. 9, no. 1, pp. 281–285, May 2023, doi: 10.30574/ijsra.2023.9.1.0410.

[2] R. Mu and X. Zeng, "A Review of Deep Learning Research," *KSII Transactions on Internet and Information Systems*, vol. 13, no. 4, pp. 1738–1764, Apr. 2019, doi: 10.3837/tiis.2019.04.001.

[3] X. Wang, "The application of NLP in information retrieval," *Applied and Computational Engineering*, vol. 42, no. 1, pp. 290–297, Feb. 2024, doi: 10.54254/2755-2721/42/20230795.

[4] H. Li, G. K. Rajbahadur, and C.-P. Bezemer, "Studying the Impact of TensorFlow and PyTorch Bindings on Machine Learning Software Quality," *ACM Transactions on Software Engineering and Methodology*, Jul. 2024, doi: 10.1145/3678168.

[5] "ETHICAL CONSIDERATION IN AI," *International Research Journal of Modernization in Engineering Technology and Science*, May 2024, doi: 10.56726/IRJMETS55881.

[6] H. W. Marar, "Advancements in software engineering using AI," *Computer Software and Media Applications*, vol. 6, no. 1, p. 3906, Feb. 2024, doi: 10.24294/csma.v6i1.3906.

[7] C. M. Hicks, C. S. Lee, and K. L. Foster-Marks, "The New Developer Executive Summary The New Developer AI Skill Threat, Identity Change & Developer Thriving in the Transition to AI-Assisted Software Development." [Online]. Available: https://www.pluralsight.com/product/flow/developer-success-lab/dsl-navigate-toolkit

[8] I. Ozkaya, "Application of Large Language Models to Software Engineering Tasks: Opportunities, Risks, and Implications," May 01, 2023, *IEEE Computer Society*. doi: 10.1109/MS.2023.3248401.

[9] P. Tembhekar, M. Devan, and J. Jeyaraman, "Role of GenAI in Automated Code Generation within DevOps Practices: Explore how Generative AI," *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, vol. 2, no. 2, pp. 500–512, Oct. 2023, doi: 10.60087/jklst.vol2.n2.p512.

[10] Z. Gao, "A review on statistical language and neural network based code completion," *Applied and Computational Engineering*, vol. 22, no. 1, pp. 233–239, Oct. 2023, doi: 10.54254/2755-2721/22/20231222.

[11] M. Atemkeng, S. Hamlomo, B. Welman, N. Oyetunji, P. Ataei, and J. L. K. E. Fendji, "Ethics of Software Programming with Generative AI: Is Programming without Generative AI always radical?," Aug. 2024, doi: https://doi.org/10.48550/arXiv.2408.10554.

[12] N. M. Dr. Naveenkumar Jayakumar, "Role of Machine Learning & Artificial Intelligence Techniques in Software Testing," *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 12, no. 6, pp. 2913–2921, Apr. 2021, doi: 10.17762/turcomat.v12i6.5800.

[13] M. A. Job, "Automating and Optimizing Software Testing using Artificial Intelligence Techniques," *International Journal of Advanced Computer Science and Applications*, vol. 12, no. 5, 2021, doi: 10.14569/IJACSA.2021.0120571.

[14] S. Gautam, A. Khunteta, and P. Sharma, “A Review on Software Testing Using Machine Learning Techniques,” *ECS Trans*, vol. 107, no. 1, pp. 3393–3406, Apr. 2022, doi: 10.1149/10701.3393ecst.
[15] Dusica Marijan; Arnaud Gotlieb, “Software Testing for Machine Learning,” 2022, doi: 10.48550/arxiv.2205.00210.
[16] L. Kharb, “Automated Testing in Machine Learning Systems,” *International Journal of Progressive Research in Engineering Management and Science*, Nov. 2023, doi: 10.58257/IJPREMS32282.
[17] J. Calle and C. Zapata, “QUARE: Towards a Question-Answering Model for Requirements Elicitation,” Jul. 29, 2022. doi: 10.21203/rs.3.rs-1872151/v1.
[18] C. Cheligeer, J. Huang, G. Wu, N. Bhuiyan, Y. Xu, and Y. Zeng, “Machine learning in requirements elicitation: a literature review,” *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 36, p. e32, Oct. 2022, doi: 10.1017/S0890060422000166.
[19] Zarina Che Embi, Khaleduzzaman, and Ng Kok Why, “A Systematic Review on Natural Language Processing and Machine Learning Approaches to Improve Requirements Specification in Software Requirements Engineering,” *International Journal of Membrane Science and Technology*, vol. 10, no. 2, pp. 1563–1577, Sep. 2023, doi: 10.15379/ijmst.v10i2.1828.
[20] W. Alhoshan, R. Batista-Navarro, and L. Zhao, “Towards a Corpus of Requirements Documents Enriched with Semantic Frame Annotations,” in *2018 IEEE 26th International Requirements Engineering Conference (RE)*, IEEE, Aug. 2018, pp. 428–431. doi: 10.1109/RE.2018.00055.
[21] C. Liu, Z. Zhao, L. Zhang, and Z. Li, “Automated Conditional Statements Checking for Complete Natural Language Requirements Specification,” *Applied Sciences*, vol. 11, no. 17, p. 7892, Aug. 2021, doi: 10.3390/app11177892.
[22] Q. Lu, L. Zhu, J. Whittle, and J. B. Michael, “Software Engineering for Responsible AI,” *Computer (Long Beach Calif)*, vol. 56, no. 4, pp. 13–16, Apr. 2023, doi: 10.1109/MC.2023.3242055.
[23] R. Khankhoje, “Quality Challenges and Imperatives in Smart AI Software,” in *Soft Computing, Artificial Intelligence and Applications*, Academy & Industry Research Collaboration Center, Dec. 2023, pp. 143–154. doi: 10.5121/csit.2023.132412.
[24] A. Barredo Arrieta *et al.*, “Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI,” *Information Fusion*, vol. 58, pp. 82–115, Jun. 2020, doi: 10.1016/j.inffus.2019.12.012.
[25] E. A. Abdelnabi, A. M. Maatuk, and M. Hagal, “Generating UML Class Diagram from Natural Language Requirements: A Survey of Approaches and Techniques,” in *2021 IEEE 1st International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering MI-STA*, IEEE, May 2021, pp. 288–293. doi: 10.1109/MI-STA52233.2021.9464433.
[26] O. MacMillan-Scott and M. Musolesi, “(Ir)rationality and cognitive biases in large language models,” *R Soc Open Sci*, vol. 11, no. 6, Jun. 2024, doi: 10.1098/rsos.240255.
[27] R. Dhar, K. Vaidhyanathan, and V. Varma, “Can LLMs Generate Architectural Design Decisions? -An Exploratory Empirical study,” Mar. 2024, [Online]. Available: http://arxiv.org/abs/2403.01709

[28] M. Tsilimigkounakis, “Exploring the utilization of LLM tools in Software Architecture,” Athens, Nov. 2024.
[29] T. Eisenreich, S. Speth, and S. Wagner, “From Requirements to Architecture: An AI-Based Journey to Semi-Automatically Generate Software Architectures,” Jan. 2024, doi: 10.1145/3643660.3643942.
[30] S. Yang and H. Sahraoui, “Towards automatically extracting UML class diagrams from natural language specifications,” in *Proceedings - ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022: Companion Proceedings*, Association for Computing Machinery, Inc, Oct. 2022, pp. 396–403. doi: 10.1145/3550356.3561592.
[31] I. Altawaiha and A. Al-Hgaish, “ClassDiagGen Tool: Fine-Tuning the GPT-3 Model for Auto- mated Class Diagram Generation from Textual Descriptions,” May 02, 2024. doi: 10.21203/rs.3.rs-4350615/v1.
[32] Ian. Sommerville, *Software engineering*. Pearson, 2011.
[33] P. Kumar and Y. Singh, “A Software Reliability Growth Model for Three-Tier Client Server System.”
[34] M. Yener and A. Theedom, “Model View Controller Pattern,” 2015. [Online]. Available: www.wrox.com/go/
[35] T. Engel, M. Langermeier, B. Bauer, and A. Hofmann, “Evaluation of microservice architectures: A metric and tool-based approach,” in *Lecture Notes in Business Information Processing*, Springer Verlag, 2018, pp. 74–89. doi: 10.1007/978-3-319-92901-9_8.
[36] J. Bogner, S. Wagner, and A. Zimmermann, “Towards a practical maintainability quality model for serviceand microservice-based systems,” in *ACM International Conference Proceeding Series*, Association for Computing Machinery, Sep. 2017, pp. 195–198. doi: 10.1145/3129790.3129816.
[37] J. Bogner, S. Wagner, and A. Zimmermann, “Automatically measuring the maintainability of service- and microservice-based systems - a literature review,” in *ACM International Conference Proceeding Series*, Association for Computing Machinery, Oct. 2017, pp. 107–115. doi: 10.1145/3143434.3143443.
[38] C. Richardson, “Microservices Patterns with Examples in JAVA,” 2019.
[39] Malhotra Nishant, “Microservices Design Patterns,” 2023. [Online]. Available: www.valuelabs.com